
RT-THREAD 编程指南

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 @2019



WWW.RT-THREAD.ORG

Thursday 1st October, 2020

版本和修订

Date	Version	Author	Note
2013-05-14	v1.0.0	bernard	初始版本
2018-12-29	v2.0.0	yangjie	内核内容修订
2018-12.29	v2.0.0	misonyo	组件及设备内容修订

目录

版本和修订	i
目录	ii
1 RT-Thread 简介	1
1.1 RT-Thread 概述	1
1.2 许可协议	1
1.3 RT-Thread 的架构	2
2 内核基础	4
2.1 RT-Thread 内核介绍	4
2.1.1 线程调度	5
2.1.2 时钟管理	5
2.1.3 线程间同步	5
2.1.4 线程间通信	5
2.1.5 内存管理	6
2.1.6 I/O 设备管理	6
2.2 RT-Thread 启动流程	6
2.3 RT-Thread 程序内存分布	8
2.4 RT-Thread 自动初始化机制	10
2.5 RT-Thread 内核对象模型	11
2.5.1 静态对象和动态对象	11
2.5.2 内核对象管理架构	13
2.5.3 对象控制块	15
2.5.4 内核对象管理方式	16
2.5.4.1 初始化对象	16
2.5.4.2 脱离对象	17

2.5.4.3 分配对象	17
2.5.4.4 删除对象	17
2.5.4.5 辨别对象	18
2.6 RT-Thread 内核配置示例	18
2.7 常见宏定义说明	21
3 线程管理	23
3.1 线程管理的功能特点	24
3.2 线程的工作机制	24
3.2.1 线程控制块	24
3.2.2 线程重要属性	25
3.2.2.1 线程栈	25
3.2.2.2 线程状态	26
3.2.2.3 线程优先级	27
3.2.2.4 时间片	27
3.2.2.5 线程的入口函数	27
3.2.2.6 线程错误码	28
3.2.3 线程状态切换	28
3.2.4 系统线程	29
3.2.4.1 空闲线程	29
3.2.4.2 主线程	29
3.3 线程的管理方式	30
3.3.1 创建和删除线程	30
3.3.2 初始化和脱离线程	31
3.3.3 启动线程	33
3.3.4 获得当前线程	33
3.3.5 使线程让出处理器资源	33
3.3.6 使线程睡眠	34
3.3.7 挂起和恢复线程	34
3.3.8 控制线程	35
3.3.9 设置和删除空闲钩子	36
3.3.10 设置调度器钩子	36
3.4 线程应用示例	37

3.4.1	创建线程示例	37
3.4.2	线程时间片轮转调度示例	39
3.4.3	线程调度器钩子示例	41
4	时钟管理	44
4.1	时钟节拍	44
4.1.1	时钟节拍的实现方式	44
4.1.2	获取时钟节拍	45
4.2	定时器管理	46
4.2.1	RT-Thread 定时器介绍	46
4.2.1.1	HARD_TIMER 模式	46
4.2.1.2	SOFT_TIMER 模式	47
4.2.2	定时器工作机制	47
4.2.2.1	定时器控制块	48
4.2.2.2	定时器跳表 (Skip List) 算法	48
4.2.3	定时器的管理方式	49
4.2.3.1	创建和删除定时器	50
4.2.3.2	初始化和脱离定时器	51
4.2.3.3	启动和停止定时器	52
4.2.3.4	控制定时器	53
4.3	定时器应用示例	54
4.4	高精度延时	57
5	线程间同步	59
5.1	信号量	60
5.1.1	信号量工作机制	60
5.1.2	信号量控制块	61
5.1.3	信号量的管理方式	61
5.1.3.1	创建和删除信号量	61
5.1.3.2	初始化和脱离信号量	62
5.1.3.3	获取信号量	63
5.1.3.4	无等待获取信号量	64
5.1.3.5	释放信号量	64
5.1.4	信号量应用示例	65

5.1.5	信号量的使用场合	71
5.1.5.1	线程同步	71
5.1.5.2	锁	72
5.1.5.3	中断与线程的同步	72
5.1.5.4	资源计数	73
5.2	互斥量	73
5.2.1	互斥量工作机制	73
5.2.2	互斥量控制块	75
5.2.3	互斥量的管理方式	75
5.2.3.1	创建和删除互斥量	75
5.2.3.2	初始化和脱离互斥量	76
5.2.3.3	获取互斥量	77
5.2.3.4	释放互斥量	78
5.2.4	互斥量应用示例	78
5.2.5	互斥量的使用场合	84
5.3	事件集	84
5.3.1	事件集工作机制	84
5.3.2	事件集控制块	85
5.3.3	事件集的管理方式	85
5.3.3.1	创建和删除事件集	86
5.3.3.2	初始化和脱离事件集	87
5.3.3.3	发送事件	88
5.3.3.4	接收事件	88
5.3.4	事件集应用示例	89
5.3.5	事件集的使用场合	92
6	线程间通信	93
6.1	邮箱	93
6.1.1	邮箱的工作机制	93
6.1.2	邮箱控制块	94
6.1.3	邮箱的管理方式	94
6.1.3.1	创建和删除邮箱	95
6.1.3.2	初始化和脱离邮箱	96

6.1.3.3	发送邮件	97
6.1.3.4	等待方式发送邮件	97
6.1.3.5	接收邮件	98
6.1.4	邮箱使用示例	98
6.1.5	邮箱的使用场合	101
6.2	消息队列	102
6.2.1	消息队列的工作机制	102
6.2.2	消息队列控制块	103
6.2.3	消息队列的管理方式	103
6.2.3.1	创建和删除消息队列	104
6.2.3.2	初始化和脱离消息队列	105
6.2.3.3	发送消息	106
6.2.3.4	等待方式发送消息	106
6.2.3.5	发送紧急消息	107
6.2.3.6	接收消息	108
6.2.4	消息队列应用示例	108
6.2.5	消息队列的使用场合	112
6.2.5.1	发送消息	112
6.2.5.2	同步消息	113
6.3	信号	114
6.3.1	信号的工作机制	114
6.3.2	信号的管理方式	114
6.3.2.1	安装信号	115
6.3.2.2	阻塞信号	115
6.3.2.3	解除信号阻塞	116
6.3.2.4	发送信号	116
6.3.2.5	等待信号	116
6.3.3	信号应用示例	117
7	内存管理	120
7.1	内存管理的功能特点	120
7.2	内存堆管理	121
7.2.1	小内存管理算法	122

7.2.2	slab 管理算法	123
7.2.3	memheap 管理算法	124
7.2.4	内存堆配置和初始化	125
7.2.5	内存堆的管理方式	126
7.2.5.1	分配和释放内存块	126
7.2.5.2	重分配内存块	127
7.2.5.3	分配多内存块	127
7.2.5.4	设置内存钩子函数	128
7.2.6	内存堆管理应用示例	129
7.3	内存池	130
7.3.1	内存池工作机制	131
7.3.1.1	内存池控制块	131
7.3.1.2	内存块分配机制	132
7.3.2	内存池的管理方式	133
7.3.2.1	创建和删除内存池	134
7.3.2.2	初始化和脱离内存池	135
7.3.2.3	分配和释放内存块	136
7.3.3	内存池应用示例	137
8	中断管理	140
8.1	Cortex-M CPU 架构基础	141
8.1.1	寄存器简介	141
8.1.2	操作模式和特权级别	143
8.1.3	嵌套向量中断控制器	143
8.1.4	PendSV 系统调用	144
8.2	RT-Thread 中断工作机制	144
8.2.1	中断向量表	144
8.2.2	中断处理过程	146
8.2.2.1	中断前导程序	146
8.2.2.2	用户中断服务程序	147
8.2.2.3	中断后续程序	148
8.2.3	中断嵌套	149
8.2.4	中断栈	150

8.2.5 中断的底半处理	150
8.3 RT-Thread 中断管理接口	152
8.3.1 中断服务程序挂接	152
8.3.2 中断源管理	153
8.3.3 全局中断开关	154
8.3.4 中断通知	156
8.4 中断与轮询	156
8.5 全局中断开关使用示例	158
9 内核移植	160
9.1 CPU 架构移植	160
9.1.1 实现全局中断开关	161
9.1.1.1 关闭全局中断	161
9.1.1.2 打开全局中断	162
9.1.2 实现线程栈初始化	162
9.1.3 实现上下文切换	164
9.1.3.1 实现 rt_hw_context_switch_to()	166
9.1.3.2 实现 rt_hw_context_switch0/ rt_hw_context_switch_interrupt0	167
9.1.3.3 实现 PendSV 中断	169
9.1.4 实现时钟节拍	172
9.2 BSP 移植	172
10 Env 用户手册	173
10.1 主要特性	173
10.2 准备工作	173
10.3 Env 的使用方法	173
10.3.1 打开 Env 控制台	174
10.3.1.1 方法一：点击 Env 目录下可执行文件	174
10.3.1.2 方法二：在文件夹中通过右键菜单打开 Env 控制台	174
10.3.2 编译 BSP	176
10.3.2.1 第一步：切换到 BSP 根目录	176
10.3.2.2 第二步：bsp 的编译	177
10.3.3 BSP 配置：menuconfig	179
10.3.3.1 快捷键介绍	179

10.3.3.2 修改配置	179
10.3.3.3 保存配置	180
10.3.4 软件包管理: package	180
10.3.4.1 下载、更新、删除软件包	180
10.3.4.2 升级本地软件包信息	181
10.3.5 Env 工具配置	181
10.4 在项目中使用 Env	183
10.4.1 使用 Env 的要求	183
10.4.2 menuconfig 中选项的修改方法	183
10.4.3 新的项目添加 menuconfig 功能	183
10.4.4 旧项目添加 menuconfig 功能	183
10.4.5 用户软件包管理功能	184
10.5 使用 pip 扩展更多功能	184
10.6 Env 工具使用注意事项	184
10.7 常见问题	185
10.7.1 Q: Env 工具出现乱码怎么办?	185
10.7.2 Q: 提示找不到 git 命令?	185
10.7.3 Q: 提示找不到 CMD 命令?	185
10.7.4 Q: 运行 python 的时候提示 no module named site 怎么办?	185
10.7.5 Q: 在 Env 下能生成哪些类型的工程?	185
10.7.6 Q: 自己制作的 BSP 如何能支持 menuconfig?	185
10.7.7 Q: pkgs -upgrade 命令和 pkgs -update 命令有什么区别?	186
10.7.8 Q: VC98 文件夹问题	186
10.7.9 Q: 使用 menuconfig 命令提示 “can't find file Kconfig”。	187
10.7.10Q: IOError: [Errno 2] No such file or directory: ‘nul’	187
10.8 常用资料链接	187
11 I/O 设备模型	188
11.1 I/O 设备介绍	188
11.1.1 I/O 设备模型框架	188
11.1.2 I/O 设备模型	190
11.1.3 I/O 设备类型	191
11.2 创建和注册 I/O 设备	193

11.3 访问 I/O 设备	196
11.3.1 查找设备	197
11.3.2 初始化设备	197
11.3.3 打开和关闭设备	198
11.3.4 控制设备	199
11.3.5 读写设备	199
11.3.6 数据收发回调	200
11.3.7 设备访问示例	201
12 UART 设备	203
12.1 UART 简介	203
12.2 访问串口设备	204
12.2.1 查找串口设备	204
12.2.2 打开串口设备	204
12.2.3 控制串口设备	206
12.2.4 发送数据	208
12.2.5 设置发送完成回调函数	209
12.2.6 设置接收回调函数	209
12.2.7 接收数据	211
12.2.8 关闭串口设备	212
12.3 串口设备使用示例	212
12.3.1 中断接收及轮询发送	212
12.3.2 DMA 接收及轮询发送	215
12.3.3 串口接收不定长数据	219
13 PIN 设备	222
13.1 引脚简介	222
13.2 访问 PIN 设备	223
13.2.1 获取引脚编号	223
13.2.1.1 使用宏定义	223
13.2.1.2 查看驱动文件	223
13.2.2 设置引脚模式	224
13.2.3 设置引脚电平	224
13.2.4 读取引脚电平	225

13.2.5 绑定引脚中断回调函数	226
13.2.6 使能引脚中断	227
13.2.7 脱离引脚中断回调函数	227
13.3 PIN 设备使用示例	228
14 ADC 设备	231
14.1 ADC 简介	231
14.1.1 转换过程	231
14.1.2 分辨率	232
14.1.3 精度	232
14.1.4 转换速率	232
14.2 访问 ADC 设备	232
14.2.1 查找 ADC 设备	232
14.2.2 使能 ADC 通道	233
14.2.3 读取 ADC 通道采样值	233
14.2.4 关闭 ADC 通道	234
14.2.5 FinSH 命令	235
14.3 ADC 设备使用示例	236
14.4 常见问题	237
14.4.1 Q: menuconfig 找不到 ADC 设备的配置选项?	237
15 HWTIMER 设备	238
15.1 定时器简介	238
15.2 访问硬件定时器设备	238
15.2.1 查找定时器设备	239
15.2.2 打开定时器设备	239
15.2.3 设置超时回调函数	240
15.2.4 控制定时器设备	240
15.2.5 设置定时器超时值	242
15.2.6 获取定时器当前值	243
15.2.7 关闭定时器设备	244
15.3 硬件定时器设备使用示例	245

16 I2C 总线设备	247
16.1 I2C 简介	247
16.2 访问 I2C 总线设备	248
16.2.1 查找 I2C 总线设备	249
16.2.2 数据传输	249
16.3 I2C 总线设备使用示例	251
17 PWM 设备	255
17.1 PWM 简介	255
17.2 访问 PWM 设备	256
17.2.1 查找 PWM 设备	256
17.2.2 设置 PWM 周期和脉冲宽度	257
17.2.3 使能 PWM 设备	258
17.2.4 关闭 PWM 设备通道	259
17.3 FinSH 命令	259
17.4 PWM 设备使用示例	260
18 RTC 设备	262
18.1 RTC 简介	262
18.2 访问 RTC 设备	262
18.2.1 设置日期	262
18.2.2 设置时间	263
18.2.3 获取当前时间	263
18.3 功能配置	264
18.3.1 启用 Soft RTC (软件模拟 RTC)	264
18.3.2 启用 NTP 时间自动同步	264
18.4 FinSH 命令	265
18.5 RTC 设备使用示例	265
19 SPI 设备	267
19.1 SPI 简介	267
19.2 挂载 SPI 设备	269
19.3 配置 SPI 设备	270
19.4 配置 QSPI 设备	271
19.5 访问 SPI 设备	272

19.5.1 查找 SPI 设备	272
19.5.2 自定义传输数据	273
19.5.3 传输一次数据	274
19.5.4 发送一次数据	275
19.5.5 接收一次数据	276
19.5.6 连续两次发送数据	277
19.5.7 先发送后接收数据	278
19.6 访问 QSPI 设备	279
19.6.1 传输数据	279
19.6.2 接收数据	280
19.6.3 发送数据	280
19.7 特殊使用场景	281
19.7.1 获取总线	281
19.7.2 选中片选	281
19.7.3 增加一条消息	282
19.7.4 释放片选	282
19.7.5 释放总线	282
19.8 SPI 设备使用示例	283
20 WATCHDOG 设备	285
20.1 WATCHDOG 简介	285
20.2 访问看门狗设备	285
20.2.1 查看看门狗	285
20.2.2 初始化看门狗	286
20.2.3 控制看门狗	287
20.2.4 关闭看门狗	288
20.3 看门狗设备使用示例	288
21 WLAN 设备	291
21.1 WLAN 框架简介	291
21.1.1 功能简介	292
21.1.2 配置选项	292
21.2 WLAN 初始化	293
21.2.1 连接管理初始化	294

21.2.2 设置设备模式	294
21.2.3 获取设备模式	295
21.3 WLAN 连接	295
21.3.1 连接热点	295
21.3.2 无阻塞连接	296
21.3.3 断开热点	297
21.3.4 获取连接标志	297
21.3.5 获取就绪标志	298
21.3.6 获取连接信息	298
21.3.7 获取信号强度	298
21.4 WLAN 扫描	299
21.4.1 异步扫描	299
21.4.2 同步扫描	299
21.4.3 条件扫描	300
21.4.4 获取热点个数	301
21.4.5 拷贝热点信息	301
21.4.6 获取扫描缓存	301
21.4.7 清理扫描缓存	302
21.4.8 查找最佳热点	302
21.5 WLAN 热点	302
21.5.1 启动热点	303
21.5.2 非阻塞启动热点	303
21.5.3 获取启动标志	303
21.5.4 停止热点	304
21.5.5 获取热点信息	304
21.6 WLAN 自动重连	304
21.6.1 启动/停止自动重连	304
21.6.2 获取自动重连模式	305
21.7 WLAN 事件回调	305
21.7.1 事件注册	305
21.7.2 解除注册	306
21.8 WLAN 功耗管理	307
21.8.1 设置功耗等级	307

21.8.2 获取功耗等级	307
21.9 FinSH 命令	308
21.9.1 WiFi 扫描	308
21.9.2 WiFi 连接	308
21.9.3 WiFi 断开	308
21.10 WLAN 设备使用示例	309
21.10.1扫描示例	309
21.10.2连接与断开示例	310
21.10.3自动连接示例	313
22 FinSH 控制台	316
22.1 FinSH 简介	316
22.1.1 传统命令行模式	318
22.1.2 C 语言解释器模式	318
22.2 FinSH 内置命令	318
22.2.1 显示线程状态	319
22.2.2 显示信号量状态	319
22.2.3 显示事件状态	320
22.2.4 显示互斥量状态	320
22.2.5 显示邮箱状态	321
22.2.6 显示消息队列状态	321
22.2.7 显示内存池状态	321
22.2.8 显示定时器状态	322
22.2.9 显示设备状态	322
22.2.10显示动态内存状态	323
22.3 自定义 FinSH 命令	323
22.3.1 自定义 msh 命令	323
22.3.2 自定义 C-Style 命令和变量	324
22.3.3 自定义命令重命名	325
22.4 FinSH 功能配置	325
22.5 FinSH 应用示例	327
22.5.1 不带参数的 msh 命令示例	327
22.5.2 带参数的 msh 命令示例	327
22.6 FinSH 移植	328

23 虚拟文件系统	330
23.1 DFS 简介	330
23.1.1 DFS 架构	331
23.1.2 POSIX 接口层	332
23.1.3 虚拟文件系统层	332
23.1.4 设备抽象层	333
23.2 挂载管理	333
23.2.1 初始化 DFS 组件	333
23.2.2 注册文件系统	334
23.2.3 将存储设备注册为块设备	334
23.2.4 格式化文件系统	335
23.2.5 挂载文件系统	336
23.2.6 卸载文件系统	337
23.3 文件管理	337
23.3.1 打开和关闭文件	337
23.3.2 读写数据	338
23.3.3 重命名	339
23.3.4 取得状态	340
23.3.5 删除文件	340
23.3.6 同步文件数据到存储设备	340
23.3.7 查询文件系统相关信息	341
23.3.8 监视 I/O 设备状态	341
23.4 目录管理	342
23.4.1 创建和删除目录	342
23.4.2 打开和关闭目录	343
23.4.3 读取目录	344
23.4.4 取得目录流的读取位置	344
23.4.5 设置下次读取目录的位置	344
23.4.6 重设读取目录的位置为开头位置	345
23.5 DFS 配置选项	345
23.5.1 elm-FatFs 文件系统配置选项	346
23.5.1.1 长文件名	347
23.5.1.2 编码方式	347

23.5.1.3 文件系统扇区大小	348
23.5.1.4 可重入性	348
23.5.1.5 更多配置	348
23.6 DFS 应用示例	348
23.6.1 FinSH 命令	348
23.6.2 读写文件示例	350
23.6.3 更改文件名称示例	351
23.6.4 获取文件状态示例	351
23.6.5 创建目录示例	352
23.6.6 读取目录示例	353
23.6.7 设置读取目录位置示例	354
23.7 常见问题	356
23.7.1 Q: 发现文件名或者文件夹名称显示不正常怎么办?	356
23.7.2 Q: 文件系统初始化失败怎么办?	356
23.7.3 Q: 创建文件系统 mkfs 命令失败怎么办?	356
23.7.4 Q: 文件系统挂载失败怎么办?	356
23.7.5 Q: SFUD 探测不到 Flash 所使用的具体型号怎么办?	356
23.7.6 Q: 存储设备的 benchmark 测试耗时过长是怎么回事?	357
23.7.7 Q: SPI Flash 实现 elmfat 文件系统, 如何保留部分扇区不被文件系统使用?	357
23.7.8 Q: 测试文件系统过程中程序卡住了怎么办?	357
23.7.9 Q: 如何一步步检查文件系统出现的问题?	357
24 AT 组件	358
24.1 AT 命令简介	358
24.2 AT 组件简介	359
24.3 AT Server	360
24.3.1 AT Server 配置	360
24.3.2 AT Server 初始化	361
24.3.3 自定义 AT 命令添加方式	361
24.3.4 AT Server API 接口	363
24.3.4.1 发送数据至客户端（不换行）	363
24.3.4.2 发送数据至客户端（换行）	363
24.3.4.3 发送命令执行结果至客户端	363

24.3.4.4 解析输入命令参数	364
24.4 AT Client	366
24.4.1 AT Client 配置	366
24.4.2 AT Client 初始化	367
24.4.3 AT Client 数据收发方式	367
24.4.3.1 创建响应结构体	368
24.4.3.2 删除响应结构体	368
24.4.3.3 设置响应结构体参数	368
24.4.3.4 发送命令并接收响应	369
24.4.4 AT Client 数据解析方式	371
24.4.4.1 获取指定行号的响应数据	371
24.4.4.2 获取指定关键字的响应数据	371
24.4.4.3 解析指定行号的响应数据	371
24.4.4.4 解析指定关键字行的响应数据	372
24.4.4.5 串口配置信息解析示例	373
24.4.4.6 IP 和 MAC 地址解析示例	373
24.4.5 AT Client URC 数据处理	374
24.4.5.1 URC 数据列表初始化	374
24.4.6 AT Client 其他 API 接口介绍	375
24.4.6.1 发送指定长度数据	375
24.4.6.2 接收指定长度数据	376
24.4.6.3 设置接收数据的行结束符	376
24.4.6.4 等待模块初始化完成	376
24.4.7 AT Client 多客户端支持	377
24.5 常见问题	379
24.5.1 Q: 开启 AT 命令收发数据实时打印功能, shell 上日志显示错误怎么办?	379
24.5.2 Q: AT Socket 功能启动时, 编译提示 “The AT socket device is not selected, please select it through the env menuconfig”?	379
24.5.3 Q: AT Socket 功能数据接收超时或者数据接收不全?	379
25 SAL 套接字抽象层	380
25.1 SAL 简介	380
25.1.1 SAL 网络框架	380
25.1.2 工作原理	381

25.1.2.1 多协议栈接入与接口函数统一抽象功能	382
25.1.2.2 SAL TLS 加密传输功能	383
25.1.3 配置选项	386
25.2 初始化	386
25.3 BSD Socket API 介绍	386
25.3.1 创建套接字 (socket)	387
25.3.2 绑定套接字 (bind)	387
25.3.3 监听套接字 (listen)	389
25.3.4 接收连接 (accept)	390
25.3.5 建立连接 (connect)	390
25.3.6 TCP 数据发送 (send)	390
25.3.7 TCP 数据接收 (recv)	391
25.3.8 UDP 数据发送 (sendto)	391
25.3.9 UDP 数据接收 (recvfrom)	392
25.3.10 关闭套接字 (closesocket)	392
25.3.11 按设置关闭套接字 (shutdown)	393
25.3.12 设置套接字选项 (setsockopt)	393
25.3.13 获取套接字选项 (getsockopt)	394
25.3.14 获取远端地址信息 (getpeername)	394
25.3.15 获取本地地址信息 (getsockname)	395
25.3.16 配置套接字参数 (ioctlsocket)	395
25.4 网络协议栈接入方式	396
26 ulog 日志	400
26.1 ulog 简介	400
26.1.1 ulog 架构	401
26.1.2 配置选项	401
26.1.3 日志级别	402
26.1.4 日志标签	403
26.2 日志初始化	403
26.2.1 初始化	403
26.2.2 去初始化	404
26.3 日志输出 API	404

26.4 日志使用示例	405
26.4.1 使用示例	405
26.4.2 在中断 ISR 中使用	406
26.4.3 设置日志格式	407
26.4.4 hexdump 输出使用	409
26.5 日志高级功能	410
26.5.1 日志后端	411
26.5.1.1 注册后端设备	411
26.5.1.2 注销后端设备	412
26.5.1.3 后端实现及注册示例	412
26.5.2 异步日志	413
26.5.2.1 配置选项	414
26.5.2.2 使用示例	415
26.5.3 日志动态过滤器	416
26.5.3.1 按模块的级别过滤	416
26.5.3.2 按标签全局过滤	417
26.5.3.3 按级别全局过滤	417
26.5.3.4 按关键词全局过滤	418
26.5.3.5 查看过滤器信息	418
26.5.3.6 使用示例	419
26.5.4 系统异常时的使用	422
26.5.4.1 断言	422
26.5.4.2 CmBacktrace	423
26.5.5 syslog 模式	423
26.5.5.1 日志格式	423
26.5.5.2 使用方法	424
26.5.6 从 rt_dbg.h 或 elog 迁移到 ulog	424
26.5.6.1 从 rt_dbg.h 迁移	425
26.5.6.2 从 elog (EasyLogger) 迁移	425
26.5.7 日志使用技巧	425
26.5.7.1 合理利用标签分类	425
26.5.7.2 合理利用日志级别	425
26.5.7.3 避免重复性冗余日志	425

26.5.7.4 开启更多的日志格式	425
26.5.7.5 关闭不重要的日志	426
26.6 常见问题	426
26.6.1 Q: 日志代码已执行，但是无输出。	426
26.6.2 Q: 开启 ulog 后，系统运行崩溃，例如：线程堆栈溢出。	426
26.6.3 Q: 日志内容的末尾缺失。	426
26.6.4 Q: 开启时间戳以后，为什么看不到毫秒级时间。	426
26.6.5 Q: 每次 include ulog 头文件前，都要定义 LOG_TAG 及 LOG_LVL，可否简化。	426
26.6.6 Q: 运行出现警告提示：Warning: There is no enough buffer for saving async log, please increase the ULOG_ASYNC_OUTPUT_BUF_SIZE option。	426
26.6.7 Q: 编译时提示：The idle thread stack size must more than 384 when using async output by idle (ULOG_ASYNC_OUTPUT_BY_IDLE)。	427
27 utest 测试框架	428
27.1 utest 简介	428
27.1.1 测试用例定义	428
27.1.2 测试单元定义	428
27.1.3 utest 应用框图	429
27.2 utest API	429
27.2.1 assert 宏	429
27.2.2 测试单元运行宏	430
27.2.3 测试用例导出宏	430
27.2.4 测试用例 LOG 输出接口	430
27.3 配置使能	431
27.4 应用范式	431
27.5 测试用例运行要求	433
27.6 运行测试用例	433
27.6.1 测试结果分析	434
27.7 测试用例运行流程	435
27.8 注意事项	435
28 动态模块	436
28.1 功能简介	436
28.2 使用动态模块	437
28.2.1 编译固件	437

28.2.2 编译动态模块	438
28.3 RT-Thread 动态模块 API	439
28.3.1 加载动态模块	439
28.3.2 执行动态模块	440
28.3.3 退出动态模块	440
28.3.4 查找动态模块	440
28.3.5 返回动态模块	441
28.3.6 查找符号	441
28.4 标准 POSIX 动态库 libdl API	441
28.4.1 打开动态库	441
28.4.2 查找符号	442
28.4.3 关闭动态库	442
28.5 常见问题	443
28.5.1 Env 工具的相关问题请参考《Env 用户手册》.	443
28.5.2 Q: 根据文档不能成功运行动态模块。	443
28.5.3 Q: 使用 scons 命令编译工程, 提示 undefined reference to __rtmsymtab_start.	443
29 POSIX 接口	444
29.1 Pthreads 简介	444
29.1.1 在 RT-Thread 中使用 POSIX	445
29.2 线程	445
29.2.1 线程句柄	445
29.2.2 创建线程	446
29.2.2.1 创建线程示例代码	446
29.2.3 脱离线程	447
29.2.3.1 脱离线程示例代码	448
29.2.4 等待线程结束	449
29.2.4.1 等待线程结束示例代码	450
29.2.5 退出线程	452
29.2.5.1 退出线程示例代码	452
29.3 互斥锁	454
29.3.1 互斥锁控制块	454
29.3.2 初始化互斥锁	455

29.3.3 销毁互斥锁	455
29.3.4 阻塞方式对互斥锁上锁	456
29.3.5 非阻塞方式对互斥锁上锁	456
29.3.6 解锁互斥锁	456
29.3.7 互斥锁示例代码	457
29.4 条件变量	459
29.4.1 条件变量控制块	459
29.4.2 初始化条件变量	460
29.4.3 销毁条件变量	460
29.4.4 阻塞方式获取条件变量	460
29.4.5 指定阻塞时间获取条件变量	461
29.4.6 发送满足条件信号量	461
29.4.7 广播	462
29.4.8 条件变量示例代码	462
29.5 读写锁	465
29.5.1 读写锁控制块	465
29.5.2 初始化读写锁初始化	465
29.5.3 销毁读写锁	466
29.5.4 读写锁读锁定	466
29.5.4.1 阻塞方式对读写锁读锁定	466
29.5.4.2 非阻塞方式对读写锁读锁定	467
29.5.4.3 指定阻塞时间对读写锁读锁定	467
29.5.5 读写锁写锁定	468
29.5.5.1 阻塞方式对读写锁写锁定	468
29.5.5.2 非阻塞方式写锁定读写锁	468
29.5.5.3 指定阻塞时长写锁定读写锁	469
29.5.6 读写锁解锁	469
29.5.7 读写锁示例代码	470
29.6 屏障	472
29.6.1 屏障控制块	472
29.6.2 创建屏障	472
29.6.3 销毁屏障	473
29.6.4 等待屏障	473

29.6.5 屏障示例代码	473
29.7 信号量	475
29.7.1 信号量控制块	476
29.7.2 无名信号量	476
29.7.2.1 初始化无名信号量	477
29.7.2.2 销毁无名信号量	477
29.7.3 有名信号量	477
29.7.3.1 创建或打开有名信号量	477
29.7.3.2 分离有名信号量	478
29.7.3.3 关闭有名信号量	478
29.7.4 获取信号量值	479
29.7.5 阻塞方式等待信号量	479
29.7.6 非阻塞方式获取信号量	479
29.7.7 指定阻塞时间等待信号量	480
29.7.8 发送信号量	480
29.7.9 无名信号量使用示例代码	481
29.8 消息队列	483
29.8.1 消息队列控制块	483
29.8.2 创建或打开消息队列	484
29.8.3 分离消息队列	484
29.8.4 关闭消息队列	484
29.8.5 阻塞方式发送消息	485
29.8.6 指定阻塞时间发送消息	485
29.8.7 阻塞方式接受消息	486
29.8.8 指定阻塞时间接受消息	486
29.8.9 消息队列示例代码	487
29.9 线程高级编程	490
29.9.0.1 线程属性初始化及去初始化	490
29.9.0.2 线程的分离状态	491
29.9.0.3 线程的调度策略	491
29.9.0.4 线程的调度参数	491
29.9.0.5 线程的堆栈大小	492
29.9.0.6 线程堆栈大小和地址	492

29.9.0.7 线程属性相关函数	493
29.9.0.8 线程属性示例代码	493
29.9.1 线程取消	495
29.9.1.1 发送取消请求	495
29.9.1.2 设置取消状态	495
29.9.1.3 设置取消类型	495
29.9.1.4 设置取消点	496
29.9.1.5 取消点	496
29.9.1.6 线程取消示例代码	497
29.9.2 一次性初始化	498
29.9.3 线程结束后清理	499
29.9.4 其他线程相关函数	499
29.9.4.1 判断 2 个线程是否相等	499
29.9.4.2 获取线程句柄	500
29.9.4.3 获取最大最小优先级	500
29.9.5 互斥锁属性	500
29.9.5.1 互斥锁属性初始化及去初始化	500
29.9.5.2 互斥锁作用域	500
29.9.5.3 互斥锁类型	501
29.9.6 条件变量属性	501
29.9.6.1 获取条件变量作用域	502
29.9.7 读写锁属性	502
29.9.7.1 初始化属性	502
29.9.7.2 获取作用域	502
29.9.8 屏障属性	503
29.9.8.1 初始化属性	503
29.9.8.2 获取作用域	503
29.9.9 消息队列属性	504
29.9.9.1 获取属性	504

30 电源管理组件	505
30.1 PM 组件介绍	505
30.1.1 主要特点	506
30.1.2 工作原理	506
30.2 设计架构	507
30.3 低功耗状态和模式	508
30.3.1 模式的请求和释放	509
30.3.2 对模式变化敏感的设备	509
30.4 调用流程	510
30.5 API 介绍	511
30.6 使用说明	512
30.7 移植说明	513
30.8 MSH 命令	515
30.8.1 请求休眠模式	515
30.8.2 释放休眠模式	515
30.8.3 设置运行模式	516
30.8.4 查看模式状态	516
30.9 常见问题及调试方法	516
31 SCons 构建工具	518
31.1 SCons 简介	518
31.1.1 什么是构建工具	518
31.1.2 RT-Thread 构建工具	518
31.1.3 安装 SCons	519
31.2 SCons 基本功能	519
31.2.1 SCons 基本命令	520
31.2.1.1 scons	520
31.2.1.2 scons -c	520
31.2.1.3 scons -target=XXX	520
31.2.1.4 scons -jN	521
31.2.1.5 scons -dist	521
31.2.1.6 scons -verbose	521
31.2.3 SCons 进阶	522

31.3.1 SCons 内置函数	522
31.3.1.1 GetCurrentDir()	522
31.3.1.2 Glob('*.*')	522
31.3.1.3 GetDepend(macro)	522
31.3.1.4 Split(str)	522
31.3.1.5 DefineGroup(name, src, depend, **parameters)	522
31.3.1.6 SConscript(dirs, variant_dir, duplicate)	523
31.4 SConscript 示例	523
31.4.1 SConscript 示例 1	523
31.4.2 SConscript 示例 2	524
31.4.3 SConscript 示例 3	525
31.4.4 SConscript 示例 4	526
31.5 使用 SCons 管理工程	527
31.5.1 添加应用代码	527
31.5.2 添加模块	528
31.5.3 添加库	531
31.5.4 编译器选项	532
31.5.5 RT-Thread 辅助编译脚本	534
31.5.6 SCons 更多使用	534

第 1 章

RT-Thread 简介

作为一名 RTOS 的初学者，也许你对 RT-Thread 还比较陌生。然而，随着你的深入接触，你会逐渐发现 RT-Thread 的魅力和它相较于其他同类型 RTOS 的种种优越之处。RT-Thread 是一款完全由国内团队开发维护的嵌入式实时操作系统（RTOS），具有完全的自主知识产权。经过近 12 个年头的沉淀，伴随着物联网的兴起，它正演变成一个功能强大、组件丰富的物联网操作系统。

1.1 RT-Thread 概述

RT-Thread，全称是 Real Time-Thread，顾名思义，它是一个嵌入式实时多线程操作系统，基本属性之一是支持多任务，允许多个任务同时运行并不意味着处理器在同一时刻真地执行了多个任务。事实上，一个处理器核心在某一时刻只能运行一个任务，由于每次对一个任务的执行时间很短、任务与任务之间通过任务调度器进行非常快速地切换（调度器根据优先级决定此刻该执行的任务），给人造成多个任务在一个时刻同时运行的错觉。在 RT-Thread 系统中，任务通过线程实现的，RT-Thread 中的线程调度器也就是以上提到的任务调度器。

RT-Thread 主要采用 C 语言编写，浅显易懂，方便移植。它把面向对象的设计方法应用到实时系统设计中，使得代码风格优雅、架构清晰、系统模块化并且可裁剪性非常好。针对资源受限的微控制器（MCU）系统，可通过方便易用的工具，裁剪出仅需要 3KB Flash、1.2KB RAM 内存资源的 NANO 版本（NANO 是 RT-Thread 官方于 2017 年 7 月份发布的一个极简版内核）；而对于资源丰富的物联网设备，RT-Thread 又能使用在线的软件包管理工具，配合系统配置工具实现直观快速的模块化裁剪，无缝地导入丰富的软件功能包，实现类似 Android 的图形界面及触摸滑动效果、智能语音交互效果等复杂功能。

相较于 Linux 操作系统，RT-Thread 体积小，成本低，功耗低、启动快速，除此以外 RT-Thread 还具有实时性高、占用资源小等特点，非常适合于各种资源受限（如成本、功耗限制等）的场合。虽然 32 位 MCU 是它的主要运行平台，实际上很多带有 MMU、基于 ARM9、ARM11 甚至 Cortex-A 系列级别 CPU 的应用处理器在特定应用场合也适合使用 RT-Thread。

1.2 许可协议

RT-Thread 系统完全开源，3.1.0 及以前的版本遵循 GPL V2 + 开源许可协议。从 3.1.0 以后的版本遵循 Apache License 2.0 开源许可协议，可以免费在商业产品中使用，并且不需要公开私有代码。

1.3 RT-Thread 的架构

近年来，物联网（Internet Of Things, IoT）概念广为普及，物联网市场发展迅猛，嵌入式设备的联网已是大势所趋。终端联网使得软件复杂性大幅增加，传统的 RTOS 内核已经越来越难满足市场的需求，在这种情况下，物联网操作系统（IoT OS）的概念应运而生。物联网操作系统是指以操作系统内核（可以是 RTOS、Linux 等）为基础，包括如文件系统、图形库等较为完整的中间件组件，具备低功耗、安全、通信协议支持和云端连接能力的软件平台，RT-Thread 就是一个 IoT OS。

RT-Thread 与其他很多 RTOS 如 FreeRTOS、uC/OS 的主要区别之一是，它不仅仅是一个实时内核，还具备丰富的中间层组件，如下图所示。

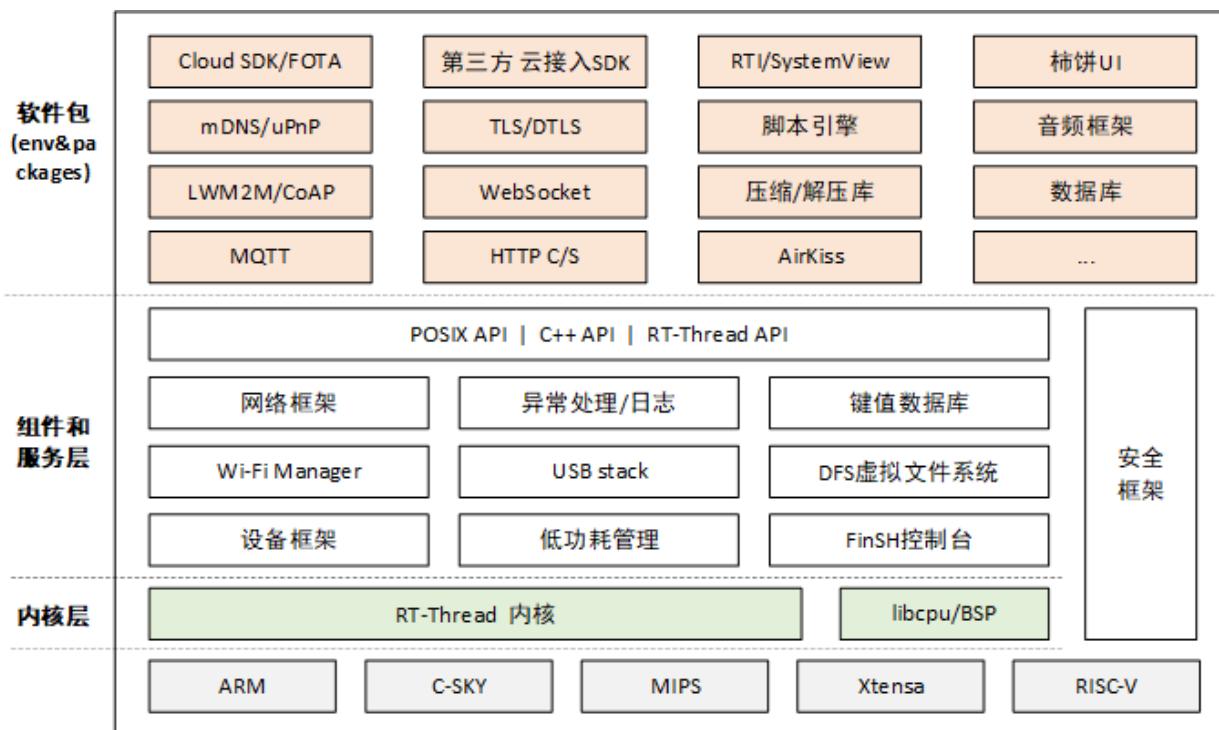


图 1.1: RT-Thread 软件框架图

它具体包括以下部分：

- **内核层：** RT-Thread 内核，是 RT-Thread 的核心部分，包括了内核系统中对象的实现，例如多线程及其调度、信号量、邮箱、消息队列、内存管理、定时器等；libcpu/BSP（芯片移植相关文件 / 板级支持包）与硬件密切相关，由外设驱动和 CPU 移植构成。
- **组件与服务层：** 组件是基于 RT-Thread 内核之上的上层软件，例如虚拟文件系统、FinSH 命令行界面、网络框架、设备框架等。采用模块化设计，做到组件内部高内聚，组件之间低耦合。
- **RT-Thread 软件包：** 运行于 RT-Thread 物联网操作系统平台上，面向不同应用领域的通用软件组件，由描述信息、源代码或库文件组成。RT-Thread 提供了开放的软件包平台，这里存放了官方提供或开发者提供的软件包，该平台为开发者提供了众多可重用软件包的选择，这也是 RT-Thread 生态的重要组成部分。软件包生态对于一个操作系统的选择至关重要，因为这些软件包具有很强的可重用性，模块化程度很高，极大的方便应用开发者在最短时间内，打造出自己想要的系统。RT-Thread 已经支持的软件包数量已经达到 60+，如下举例：

1. 物联网相关的软件包：Paho MQTT、WebClient、mongoose、WebTerminal 等等。

2. 脚本语言相关的软件包：目前支持 JerryScript、MicroPython。
3. 多媒体相关的软件包：Openmv、mupdf。
4. 工具类软件包：CmBacktrace、EasyFlash、EasyLogger、SystemView。
5. 系统相关的软件包：RTGUI、Persimmon UI、lwext4、partition、SQLite 等等。
6. 外设库与驱动类软件包：RealTek RTL8710BN SDK。
7. 其他。

第 2 章

内核基础

本章介绍 RT-Thread 内核基础，包括：内核简介、系统的启动流程及内核配置的部分内容，为后面的章节奠定基础。

RT-Thread 内核的简单介绍，从软件架构入手讲解实时内核的组成与实现，这部分给初学者引入一些 RT-Thread 内核相关的概念与基础知识，让初学者对内核有初步的了解。学完本章，读者将会对 RT-Thread 内核有基本的了解，知道内核的组成部分、系统如何启动、内存分布情况以及内核配置方法。

2.1 RT-Thread 内核介绍

内核是操作系统最基础也是最重要的部分。下图为 RT-Thread 内核架构图，内核处于硬件层之上，内核部分包括内核库、实时内核实现。

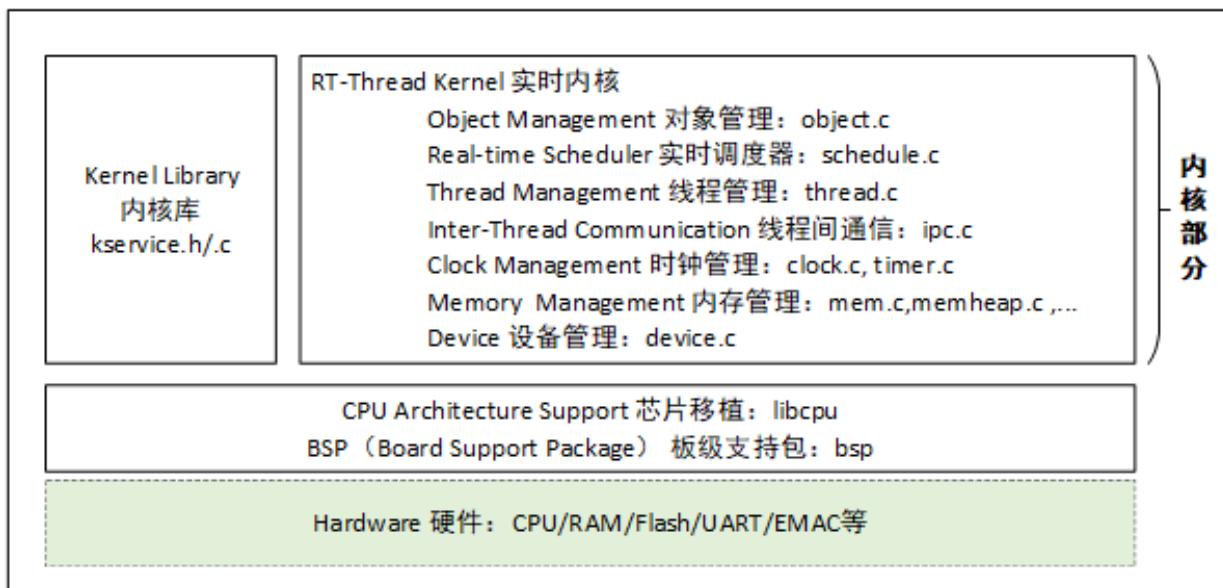


图 2.1: RT-Thread 内核及底层结构

内核库是为了保证内核能够独立运行的一套小型的类似 C 库的函数实现子集。这部分根据编译器的不同自带 C 库的情况也会有些不同，当使用 GNU GCC 编译器时，会携带更多的标准 C 库实现。

!!! tip “提示” C 库：也叫 C 运行库（C Runtime Library），它提供了类似“`strcpy`”、“`memcpy`”等函数，有些也会包括“`printf`”、“`scanf`” 函数的实现。RT-Thread Kernel Service Library 仅提供内核用到的一小部分 C 库函数实现，为了避免与标准 C 库重名，在这些函数前都会添加上 `rt_` 前缀。

实时内核的实现包括：对象管理、线程管理及调度器、线程间通信管理、时钟管理及内存管理等等，内核最小的资源占用情况是 3KB ROM, 1.2KB RAM。

2.1.1 线程调度

线程是 RT-Thread 操作系统中最小的调度单位，线程调度算法是基于优先级的全抢占式多线程调度算法，即在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。支持 256 个线程优先级（也可通过配置文件更改为最大支持 32 个或 8 个线程优先级，针对 STM32 默认配置是 32 个线程优先级），0 优先级代表最高优先级，最低优先级留给空闲线程使用；同时它也支持创建多个具有相同优先级的线程，相同优先级的线程间采用时间片的轮转调度算法进行调度，使每个线程运行相应时间；另外调度器在寻找那些处于就绪状态的具有最高优先级的线程时，所经历的时间是恒定的，系统也不限制线程数量的多少，线程数目只和硬件平台的具体内存相关。

线程管理将在《线程管理》章节详细介绍。

2.1.2 时钟管理

RT-Thread 的时钟管理以时钟节拍为基础，时钟节拍是 RT-Thread 操作系统中最小的时钟单位。RT-Thread 的定时器提供两类定时器机制：第一类是单次触发定时器，这类定时器在启动后只会触发一次定时器事件，然后定时器自动停止。第二类是周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动的停止定时器否则将永远持续执行下去。

另外，根据超时函数执行时所处的上下文环境，RT-Thread 的定时器可以设置为 HARD_TIMER 模式或者 SOFT_TIMER 模式。

通常使用定时器定时回调函数（即超时函数），完成定时服务。用户根据自己对定时处理的实时性要求选择合适类型的定时器。

定时器将在《时钟管理》章节展开讲解。

2.1.3 线程间同步

RT-Thread 采用信号量、互斥量与事件集实现线程间同步。线程通过对信号量、互斥量的获取与释放进行同步；互斥量采用优先级继承的方式解决了实时系统常见的优先级翻转问题。线程同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥量。线程通过对事件的发送与接收进行同步；事件集支持多事件的“或触发”和“与触发”，适合于线程等待多个事件的情况。

信号量、互斥量与事件集的概念将在《线程间同步》章节详细介绍。

2.1.4 线程间通信

RT-Thread 支持邮箱和消息队列等通信机制。邮箱中一封邮件的长度固定为 4 字节大小；消息队列能够接收不固定长度的消息，并把消息缓存在自己的内存空间中。邮箱效率较消息队列更为高效。邮箱和消息队列的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取。

邮箱和消息队列的概念将在《线程间通信》章节详细介绍。

2.1.5 内存管理

RT-Thread 支持静态内存池管理及动态内存堆管理。当静态内存池具有可用内存时，系统对内存块分配的时间将是恒定的；当静态内存池为空时，系统将申请内存块的线程挂起或阻塞掉（即线程等待一段时间后仍未获得内存块就放弃申请并返回，或者立刻返回。等待的时间取决于申请内存块时设置的等待时间参数），当其他线程释放内存块到内存池时，如果有挂起的待分配内存块的线程存在的话，则系统会将这个线程唤醒。

动态内存堆管理模块在系统资源不同的情况下，分别提供了面向小内存系统的内存管理算法及面向大内存系统的 SLAB 内存管理算法。

还有一种动态内存堆管理叫做 memheap，适用于系统含有多个地址可不连续的内存堆。使用 memheap 可以将多个内存堆“粘贴”在一起，让用户操作起来像是在操作一个内存堆。

内存管理的概念将在《内存管理》章节展开讲解。

2.1.6 I/O 设备管理

RT-Thread 将 PIN、I2C、SPI、USB、UART 等作为外设设备，统一通过设备注册完成。实现了按名称访问的设备管理子系统，可按照统一的 API 界面访问硬件设备。在设备驱动接口上，根据嵌入式系统的特点，对不同的设备可以挂接相应的事件。当设备事件触发时，由驱动程序通知给上层的应用程序。

I/O 设备管理的概念将在《设备模型》及《通用设备》章节展开讲解。

2.2 RT-Thread 启动流程

一般了解一份代码大多从启动部分开始，同样这里也采用这种方式，先寻找启动的源头。RT-Thread 支持多种平台和多种编译器，而 `rtthread_startup()` 函数是 RT-Thread 规定的统一启动入口。一般执行顺序是：系统先从启动文件开始运行，然后进入 RT-Thread 的启动 `rtthread_startup()`，最后进入用户入口 `main()`，如下图所示：

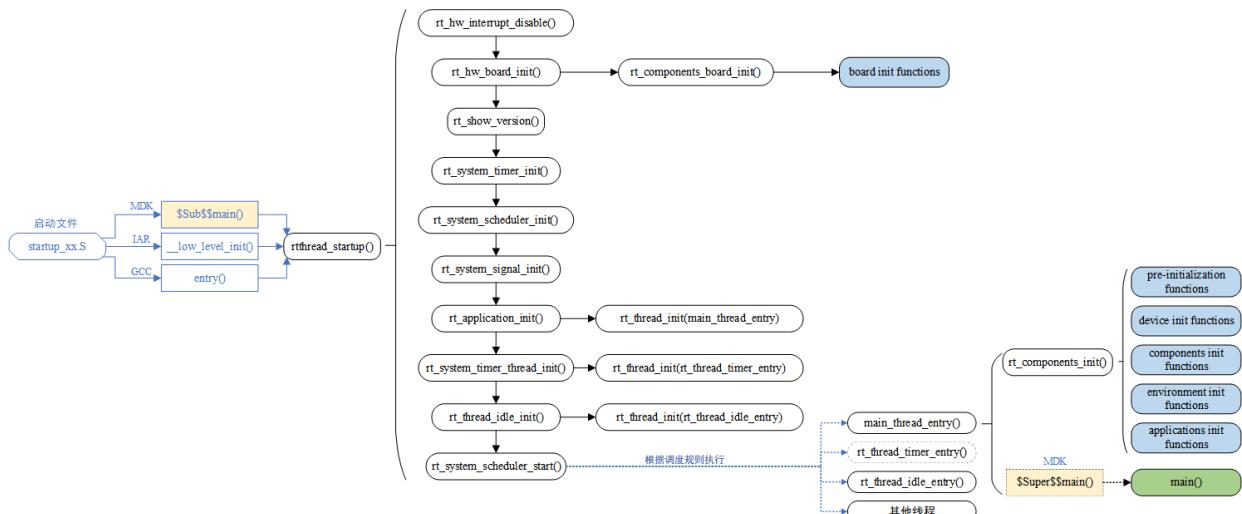


图 2.2: 启动流程

以 MDK-ARM 为例，用户程序入口为 main() 函数，位于 main.c 文件中。系统启动后先从汇编代码 startup_stm32f103xe.s 开始运行，然后跳转到 C 代码，进行 RT-Thread 系统启动，最后进入用户程序入口 main()。

为了在进入 main() 之前完成 RT-Thread 系统功能初始化，我们使用了 MDK 的扩展功能 \$Sub\$\$ 和 \$Super\$\$. 可以给 main 添加 \$Sub\$\$ 的前缀符号作为一个新功能函数 \$Sub\$\$main，这个 \$Sub\$\$main 可以先调用一些要补充在 main 之前的功能函数（这里添加 RT-Thread 系统启动，进行系统一系列初始化），再调用 \$Super\$\$main 转到 main() 函数执行，这样可以让用户不用去管 main() 之前的系统初始化操作。

关于 \$Sub\$\$ 和 \$Super\$\$ 扩展功能的使用，详见 [ARM® Compiler v5.06 for μVision® armlink User Guide](#)。

下面我们来看看在 components.c 中定义的这段代码：

```
/* $Sub$$main 函数 */
int $Sub$$main(void)
{
    rtthread_startup();
    return 0;
}
```

在这里 \$Sub\$\$main 函数调用了 rtthread_startup() 函数，其中 rtthread_startup() 函数的代码如下所示：

```
int rtthread_startup(void)
{
    rt_hw_interrupt_disable();

    /* 板级初始化：需在该函数内部进行系统堆的初始化 */
    rt_hw_board_init();

    /* 打印 RT-Thread 版本信息 */
    rt_show_version();

    /* 定时器初始化 */
    rt_system_timer_init();

    /* 调度器初始化 */
    rt_system_scheduler_init();

#ifdef RT_USING_SIGNALS
    /* 信号初始化 */
    rt_system_signal_init();
#endif

    /* 由此创建一个用户 main 线程 */
    rt_application_init();

    /* 定时器线程初始化 */
    rt_system_timer_thread_init();
```

```

/* 空闲线程初始化 */
rt_thread_idle_init();

/* 启动调度器 */
rt_system_scheduler_start();

/* 不会执行至此 */
return 0;
}

```

这部分启动代码，大致可以分为四个部分：

- (1) 初始化与系统相关的硬件；
- (2) 初始化系统内核对象，例如定时器、调度器、信号；
- (3) 创建 main 线程，在 main 线程中对各类模块依次进行初始化；
- (4) 初始化定时器线程、空闲线程，并启动调度器。

启动调度器之前，系统所创建的线程在执行 `rt_thread_startup()` 后并不会立马运行，它们会处于就绪状态等待系统调度；待启动调度器之后，系统才转入第一个线程开始运行，根据调度规则，选择的是就绪队列中优先级最高的线程。

`rt_hw_board_init()` 中完成系统时钟设置，为系统提供心跳、串口初始化，将系统输入输出终端绑定到这个串口，后续系统运行信息就会从串口打印出来。

`main()` 函数是 RT-Thread 的用户代码入口，用户可以在 `main()` 函数里添加自己的应用。

```

int main(void)
{
    /* user app entry */
    return 0;
}

```

2.3 RT-Thread 程序内存分布

一般 MCU 包含的存储空间有：片内 Flash 与片内 RAM，RAM 相当于内存，Flash 相当于硬盘。编译器会将一个程序分类为好几个部分，分别存储在 MCU 不同的存储区。

Keil 工程在编译完之后，会有相应的程序所占用的空间提示信息，如下所示：

```

linking...
Program Size: Code=48008 RO-data=5660 RW-data=604 ZI-data=2124
After Build - User command \#1: fromelf --bin.\build\rtthread-stm32.axf--output
    rtthread.bin
".\build\rtthread-stm32.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:07

```

上面提到的 Program Size 包含以下几个部分：

- 1) **Code**: 代码段，存放程序的代码部分；

- 2) RO-data: 只读数据段, 存放程序中定义的常量;
- 3) RW-data: 读写数据段, 存放初始化为非 0 值的全局变量;
- 4) ZI-data: 0 数据段, 存放未初始化的全局变量及初始化为 0 的变量;

编译完工程会生成一个 `.map` 的文件, 该文件说明了各个函数占用的尺寸和地址, 在文件的最后几行也说明了上面几个字段的关系:

```
Total RO Size (Code + RO Data) 53668 ( 52.41kB)
Total RW Size (RW Data + ZI Data) 2728 ( 2.66kB)
Total ROM Size (Code + RO Data + RW Data) 53780 ( 52.52kB)
```

- 1) RO Size 包含了 Code 及 RO-data, 表示程序占用 Flash 空间的大小;
- 2) RW Size 包含了 RW-data 及 ZI-data, 表示运行时占用的 RAM 的大小;
- 3) ROM Size 包含了 Code、RO Data 以及 RW Data, 表示烧写程序所占用的 Flash 空间的大小;

程序运行之前, 需要有文件实体被烧录到 STM32 的 Flash 中, 一般是 `bin` 或者 `hex` 文件, 该被烧录文件称为可执行映像文件。如图 3-3 中左图所示, 是可执行映像文件烧录到 STM32 后的内存分布, 它包含 RO 段和 RW 段两个部分: 其中 RO 段中保存了 Code、RO-data 的数据, RW 段保存了 RW-data 的数据, 由于 ZI-data 都是 0, 所以未包含在映像文件中。

STM32 在上电启动之后默认从 Flash 启动, 启动之后会将 RW 段中的 RW-data (初始化的全局变量) 搬运到 RAM 中, 但不会搬运 RO 段, 即 CPU 的执行代码从 Flash 中读取, 另外根据编译器给出的 ZI 地址和大小分配出 ZI 段, 并将这块 RAM 区域清零。

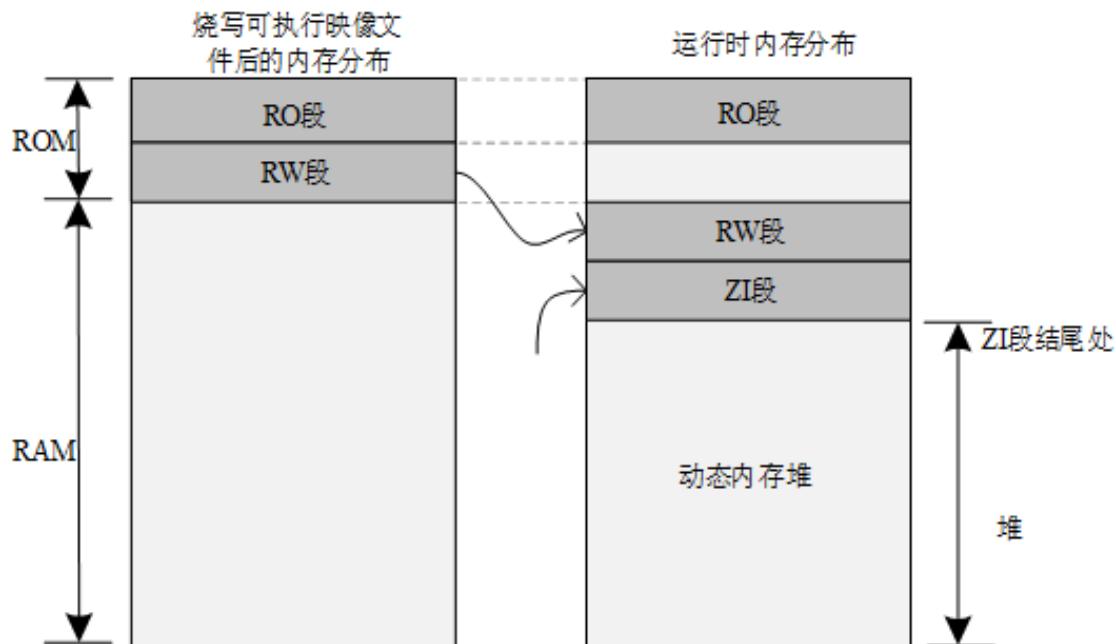


图 2.3: RT-Thread 内存分布

其中动态内存堆为未使用的 RAM 空间, 应用程序申请和释放的内存块都来自该空间。

如下面的例子:

```
rt_uint8_t* msg_ptr;
msg_ptr = (rt_uint8_t*) rt_malloc (128);
rt_memset(msg_ptr, 0, 128);
```

代码中的 `msg_ptr` 指针指向的 128 字节内存空间位于动态内存堆空间中。

而一些全局变量则是存放于 RW 段和 ZI 段中，RW 段存放的是具有初始值的全局变量（而常量形式的全局变量则放置在 RO 段中，是只读属性的），ZI 段存放的系统未初始化的全局变量，如下面的例子：

```
#include <rtthread.h>

const static rt_uint32_t sensor_enable = 0x000000FE;
rt_uint32_t sensor_value;
rt_bool_t sensor_initiated = RT_FALSE;

void sensor_init()
{
    /* ... */
}
```

`sensor_value` 存放在 ZI 段中，系统启动后会自动初始化成零（由用户程序或编译器提供的一些库函数初始化成零）。`sensor_initiated` 变量则存放在 RW 段中，而 `sensor_enable` 存放在 RO 段中。

2.4 RT-Thread 自动初始化机制

自动初始化机制是指初始化函数不需要被显式调用，只需要在函数定义处通过宏定义的方式进行申明，就会在系统启动过程中被执行。

例如在串口驱动中调用一个宏定义告知系统初始化需要调用的函数，代码如下：

```
int rt_hw_usart_init(void) /* 串口初始化函数 */
{
    ...
    /* 注册串口 1 设备 */
    rt_hw_serial_register(&serial1, "uart1",
                          RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX,
                          uart);
    return 0;
}
INIT_BOARD_EXPORT(rt_hw_usart_init); /* 使用组件自动初始化机制 */
```

示例代码最后的 `INIT_BOARD_EXPORT(rt_hw_usart_init)` 表示使用自动初始化功能，按照这种方式，`rt_hw_usart_init()` 函数就会被系统自动调用，那么它是在哪里被调用的呢？

在系统启动流程图中，有两个函数：`rt_components_board_init()` 与 `rt_components_init()`，其后的带底色方框内部的函数表示被自动初始化的函数，其中：

- “board init functions” 为所有通过 `INIT_BOARD_EXPORT(fn)` 申明的初始化函数。

2. “pre-initialization functions” 为所有通过 INIT_PREV_EXPORT(fn) 申明的初始化函数。
3. “device init functions” 为所有通过 INIT_DEVICE_EXPORT(fn) 申明的初始化函数。
4. “components init functions” 为所有通过 INIT_COMPONENT_EXPORT(fn) 申明的初始化函数。
5. “environment init functions” 为所有通过 INIT_ENV_EXPORT(fn) 申明的初始化函数。
6. “application init functions” 为所有通过 INIT_APP_EXPORT(fn) 申明的初始化函数。

`rt_components_board_init()` 函数执行的比较早，主要初始化相关硬件环境，执行这个函数时将会遍历通过 INIT_BOARD_EXPORT(fn) 申明的初始化函数表，并调用各个函数。

`rt_components_init()` 函数会在操作系统运行起来之后创建的 main 线程里被调用执行，这个时候硬件环境和操作系统已经初始化完成，可以执行应用相关代码。`rt_components_init()` 函数会遍历通过剩下的其他几个宏申明的初始化函数表。

RT-Thread 的自动初始化机制使用了自定义 RTI 符号段，将需要在启动时进行初始化的函数指针放到了该段中，形成一张初始化函数表，在系统启动过程中会遍历该表，并调用表中的函数，达到自动初始化的目的。

用来实现自动初始化功能的宏接口定义详细描述如下表所示：

初始化顺序	宏接口	描述
1	INIT_BOARD_EXPORT(fn)	非常早期的初始化，此时调度器还未启动
2	INIT_PREV_EXPORT(fn)	主要是用于纯软件的初始化、没有太多依赖的函数
3	INIT_DEVICE_EXPORT(fn)	外设驱动初始化相关，比如网卡设备
4	INIT_COMPONENT_EXPORT(fn)	组件初始化，比如文件系统或者 LWIP
5	INIT_ENV_EXPORT(fn)	系统环境初始化，比如挂载文件系统
6	INIT_APP_EXPORT(fn)	应用初始化，比如 GUI 应用

初始化函数主动通过这些宏接口进行申明，如 INIT_BOARD_EXPORT(`rt_hw_usart_init`)，链接器会自动收集所有被申明的初始化函数，放到 RTI 符号段中，该符号段位于内存分布的 RO 段中，该 RTI 符号段中的所有函数在系统初始化时会被自动调用。

2.5 RT-Thread 内核对象模型

2.5.1 静态对象和动态对象

RT-Thread 内核采用面向对象的设计思想进行设计，系统级的基础设施都是一种内核对象，例如线程，信号量，互斥量，定时器等。内核对象分为两类：静态内核对象和动态内核对象，静态内核对象通常放在 RW 段和 ZI 段中，在系统启动后在程序中初始化；动态内核对象则是从内存堆中创建的，而后手工做初始化。

以下代码是一个关于静态线程和动态线程的例子：

```
/* 线程 1 的对象和运行时用到的栈 */
static struct rt_thread thread1;
static rt_uint8_t thread1_stack[512];

/* 线程 1 入口 */
void thread1_entry(void* parameter)
{
    int i;

    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            rt_kprintf("%d\n", i);

            /* 延时 100ms */
            rt_thread_mdelay(100);
        }
    }
}

/* 线程 2 入口 */
void thread2_entry(void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf("Thread2 count:%d\n", ++count);

        /* 延时 50ms */
        rt_thread_mdelay(50);
    }
}

/* 线程例程初始化 */
int thread_sample_init()
{
    rt_thread_t thread2_ptr;
    rt_err_t result;

    /* 初始化线程 1 */
    /* 线程的入口是 thread1_entry, 参数是 RT_NULL
     * 线程栈是 thread1_stack
     * 优先级是 200, 时间片是 10 个 OS Tick
     */
    result = rt_thread_init(&thread1,
                           "thread1",
                           thread1_entry, RT_NULL,
```

```
    &thread1_stack[0], sizeof(thread1_stack),
    200, 10);

/* 启动线程 */
if (result == RT_EOK) rt_thread_startup(&thread1);

/* 创建线程 2 */
/* 线程的入口是 thread2_entry, 参数是 RT_NULL
 * 栈空间是 512, 优先级是 250, 时间片是 25 个 OS Tick
 */
thread2_ptr = rt_thread_create("thread2",
                               thread2_entry, RT_NULL,
                               512, 250, 25);

/* 启动线程 */
if (thread2_ptr != RT_NULL) rt_thread_startup(thread2_ptr);

return 0;
}
```

在这个例子中，`thread1` 是一个静态线程对象，而 `thread2` 是一个动态线程对象。`thread1` 对象的内存空间，包括线程控制块 `thread1` 与栈空间 `thread1_stack` 都是编译时决定的，因为代码中都不存在初始值，都统一放在未初始化数据段中。`thread2` 运行中用到的空间都是动态分配的，包括线程控制块（`thread2_ptr` 指向的内容）和栈空间。

静态对象会占用 RAM 空间，不依赖于内存堆管理器，内存分配时间确定。动态对象则依赖于内存堆管理器，运行时申请 RAM 空间，当对象被删除后，占用的 RAM 空间被释放。这两种方式各有利弊，可以根据实际环境需求选择具体使用方式。

2.5.2 内核对象管理架构

RT-Thread 采用内核对象管理系统来访问 / 管理所有内核对象，内核对象包含了内核中绝大部分设施，这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。

通过这种内核对象的设计方式，RT-Thread 做到了不依赖于具体的内存分配方式，系统的灵活性得到极大的提高。

RT-Thread 内核对象包括：线程，信号量，互斥量，事件，邮箱，消息队列和定时器，内存池，设备驱动等。对象容器中包含了每类内核对象的信息，包括对象类型，大小等。对象容器给每类内核对象分配了一个链表，所有的内核对象都被链接到该链表上，如图 RT-Thread 的内核对象容器及链表如下图所示：

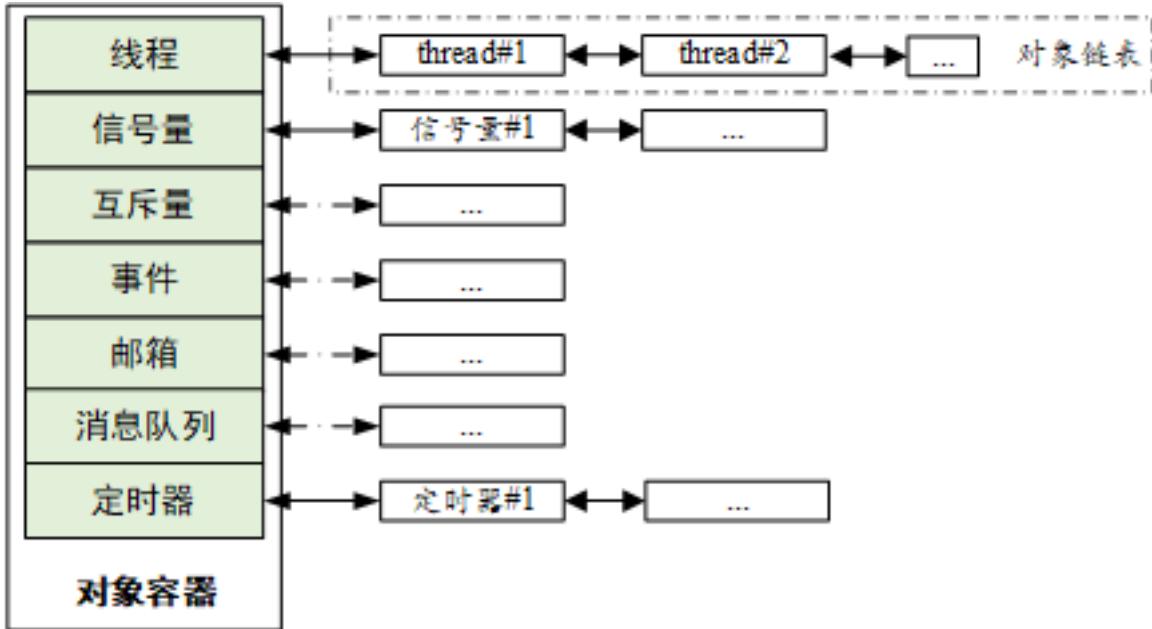


图 2.4: RT-Thread 的内核对象容器及链表

下图则显示了 RT-Thread 中各类内核对象的派生和继承关系。对于每一种具体内核对象和对象控制块，除了基本结构外，还有自己的扩展属性（私有属性），例如，对于线程控制块，在基类对象基础上进行扩展，增加了线程状态、优先级等属性。这些属性在基类对象的操作中不会用到，只有在与具体线程相关的操作中才会使用。因此从面向对象的观点，可以认为每一种具体对象是抽象对象的派生，继承了基本对象的属性并在此基础上扩展了与自己相关的属性。

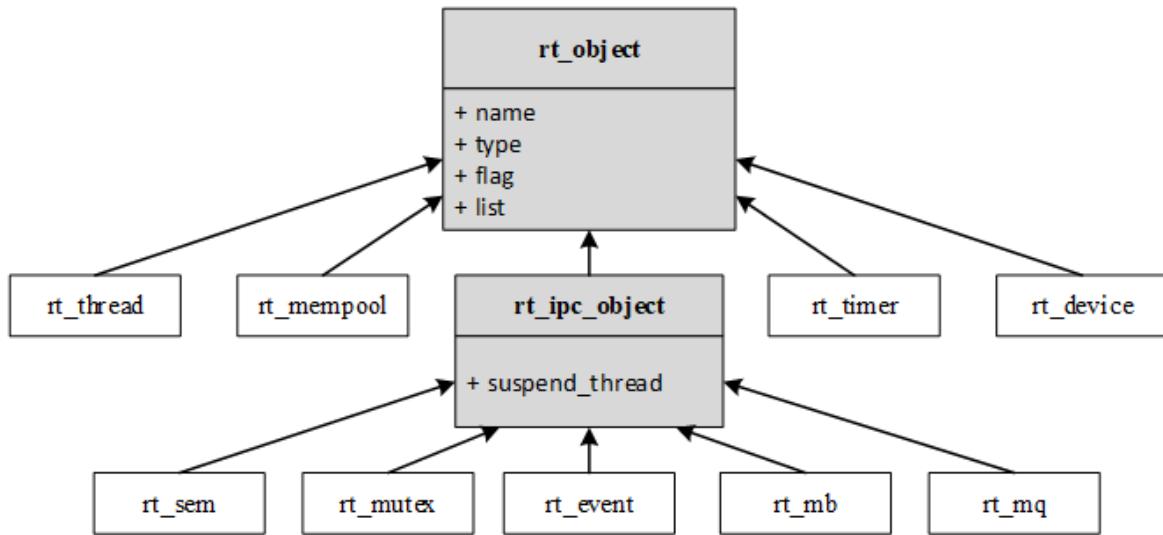


图 2.5: RT-Thread 内核对象继承关系

在对象管理模块中，定义了通用的数据结构，用来保存各种对象的共同属性，各种具体对象只需要在此基础上加上自己的某些特别的属性，就可以清楚的表示自己的特征。

这种设计方法的优点有：

- (1) 提高了系统的可重用性和扩展性，增加新的对象类别很容易，只需要继承通用对象的属性再加少

量扩展即可。

(2) 提供统一的对象操作方式，简化了各种具体对象的操作，提高了系统的可靠性。

上图中由对象控制块 `rt_object` 派生出来的有：线程对象、内存池对象、定时器对象、设备对象和 IPC 对象（IPC: Inter-Process Communication，进程间通信。在 RT-Thread 实时操作系统中，IPC 对象的作用是进行线程间同步与通信）；由 IPC 对象派生出信号量、互斥量、事件、邮箱与消息队列、信号等对象。

2.5.3 对象控制块

内核对象控制块的数据结构：

```
struct rt_object
{
    /* 内核对象名称 */
    char     name[RT_NAME_MAX];
    /* 内核对象类型 */
    rt_uint8_t type;
    /* 内核对象的参数 */
    rt_uint8_t flag;
    /* 内核对象管理链表 */
    rt_list_t list;
};
```

目前内核对象支持的类型如下：

```
enum rt_object_class_type
{
    RT_Object_Class_Thread = 0,           /* 对象为线程类型 */
#ifdef RT_USING_SEMAPHORE
    RT_Object_Class_Semaphore,          /* 对象为信号量类型 */
#endif
#ifdef RT_USING_MUTEX
    RT_Object_Class_Mutex,              /* 对象为互斥量类型 */
#endif
#ifdef RT_USING_EVENT
    RT_Object_Class_Event,              /* 对象为事件类型 */
#endif
#ifdef RT_USING_MAILBOX
    RT_Object_Class_MailBox,            /* 对象为邮箱类型 */
#endif
#ifdef RT_USING_MESSAGEQUEUE
    RT_Object_Class_MessageQueue,       /* 对象为消息队列类型 */
#endif
#ifdef RT_USING_MEMPOOL
    RT_Object_Class_MemPool,             /* 对象为内存池类型 */
#endif
#ifdef RT_USING_DEVICE
    RT_Object_Class_Device,              /* 对象为设备类型 */
#endif
};
```

```

    RT_Object_Class_Timer,           /* 对象为定时器类型 */
#define RT_USING_MODULE
    RT_Object_Class_Module,         /* 对象为模块 */
#endif
    RT_Object_Class_Unknown,        /* 对象类型未知 */
    RT_Object_Class_Static = 0x80  /* 对象为静态对象 */
};


```

从上面的类型说明，我们可以看出，如果是静态对象，那么对象类型的最高位将是 1（是 RT_Object_Class_Static 与其他对象类型的与操作），否则就是动态对象，系统最多能够容纳的对象类别数目是 127 个。

2.5.4 内核对象管理方式

内核对象容器的数据结构：

```

struct rt_object_information
{
    /* 对象类型 */
    enum rt_object_class_type type;
    /* 对象链表 */
    rt_list_t object_list;
    /* 对象大小 */
    rt_size_t object_size;
};

```

一类对象由一个 `rt_object_information` 结构体来管理，每一个这类对象的具体实例都通过链表的形式挂接在 `object_list` 上。而这一类对象的内存块尺寸由 `object_size` 标识出来（每一类对象的具体实例，他们占有的内存块大小都是相同的）。

2.5.4.1 初始化对象

在使用一个未初始化的静态对象前必须先对其进行初始化。初始化对象使用以下接口：

```

void rt_object_init(struct rt_object* object,
                    enum rt_object_class_type type,
                    const char* name)

```

当调用这个函数进行对象初始化时，系统会把这个对象放置到对象容器中进行管理，即初始化对象的一些参数，然后把这个对象节点插入到对象容器的对象链表中，对该函数的输入参数的描述如下表：

参数	描述
object	需要初始化的对象指针，它必须指向具体的对象内存块，而不能是空指针或野指针
type	对象的类型，必须是 <code>rt_object_class_type</code> 枚举类型中列出的除 <code>RT_Object_Class_Static</code> 以外的类型（对于静态对象，或使用 <code>rt_object_init</code> 接口进行初始化的对象，系统会把它标识成 <code>RT_Object_Class_Static</code> 类型）

参数	描述
name	对象的名字。每个对象可以设置一个名字，这个名字的最大长度由 RT_NAME_MAX 指定，并且系统不关心它是否是由'\0'做为终结符

2.5.4.2 脱离对象

从内核对象管理器中脱离一个对象。脱离对象使用以下接口：

```
void rt_object_detach(rt_object_t object);
```

调用该接口，可使得一个静态内核对象从内核对象容器中脱离出来，即从内核对象容器链表上删除相应的对象节点。对象脱离后，对象占用的内存并不会被释放。

2.5.4.3 分配对象

上述描述的都是对象初始化、脱离的接口，都是面向对象内存块已经有的情况下，而动态的对象则可以在需要时申请，不需要时释放出内存空间给其他应用使用。申请分配新的对象可以使用以下接口：

```
rt_object_t rt_object_allocate(enum rt_object_class_type type,
                               const char* name)
```

在调用以上接口时，系统首先需要根据对象类型来获取对象信息（特别是对象类型的大小信息以用于系统能够分配正确大小的内存数据块），而后从内存堆中分配对象所对应大小的内存空间，然后再对该对象进行必要的初始化，最后将其插入到它所在的对象容器链表中。对该函数的输入参数的描述如下表：

参数	描述
type	分配对象的类型，只能是 rt_object_class_type 中除 RT_Object_Class_Static 以外的类型。并且经过这个接口分配出来的对象类型是动态的，而不是静态的
name	对象的名字。每个对象可以设置一个名字，这个名字的最大长度由 RT_NAME_MAX 指定，并且系统不关心它是否是由'\0'做为终结符
返回	—
分配成功的对象句柄	分配成功
RT_NULL	分配失败

2.5.4.4 删除对象

对于一个动态对象，当不再使用时，可以调用如下接口删除对象，并释放相应的系统资源：

```
void rt_object_delete(rt_object_t object);
```

当调用以上接口时，首先从对象容器链表中脱离对象，然后释放对象所占用的内存。对该函数的输入参数的描述下表：

参数	描述
object	对象的句柄

2.5.4.5 辨别对象

判断指定对象是否是系统对象（静态内核对象）。辨别对象使用以下接口：

```
rt_err_t rt_object_is_systemobject(rt_object_t object);
```

调用 `rt_object_is_systemobject` 接口可判断一个对象是否是系统对象，在 RT-Thread 操作系统中，一个系统对象也就是一个静态对象，对象类型标识上 `RT_Object_Class_Static` 位置位。通常使用 `rt_object_init()` 方式初始化的对象都是系统对象。对该函数的输入参数的描述如下表：

`rt_object_is_systemobject()` 的输入参数

参数	描述
object	对象的句柄

2.6 RT-Thread 内核配置示例

RT-Thread 的一个重要特性是高度可裁剪性，支持对内核进行精细调整，对组件进行灵活拆卸。

配置主要是通过修改工程目录下的 `rtconfig.h` 文件来进行，用户可以通过打开 / 关闭该文件中的宏定义来对代码进行条件编译，最终达到系统配置和裁剪的目的，如下：

(1) RT-Thread 内核部分

```
/* 表示内核对象的名称的最大长度，若代码中对象名称的最大长度大于宏定义的长度，  
* 多余的部分将被截掉。*/  
#define RT_NAME_MAX 8  
  
/* 字节对齐时设定对齐的字节个数。常使用 ALIGN(RT_ALIGN_SIZE) 进行字节对齐。*/  
#define RT_ALIGN_SIZE 4  
  
/* 定义系统线程优先级数；通常用 RT_THREAD_PRIORITY_MAX-1 定义空闲线程的优先级 */  
#define RT_THREAD_PRIORITY_MAX 32  
  
/* 定义时钟节拍，为 100 时表示 100 个 tick 每秒，一个 tick 为 10ms */  
#define RT_TICK_PER_SECOND 100  
  
/* 检查栈是否溢出，未定义则关闭 */  
#define RT_USING_OVERFLOW_CHECK  
  
/* 定义该宏开启 debug 模式，未定义则关闭 */  
#define RT_DEBUG
```

```

/* 开启 debug 模式时：该宏定义为 0 时表示关闭打印组件初始化信息，定义为 1 时表示启用 */
#define RT_DEBUG_INIT 0
/* 开启 debug 模式时：该宏定义为 0 时表示关闭打印线程切换信息，定义为 1 时表示启用 */
#define RT_DEBUG_THREAD 0

/* 定义该宏表示开启钩子函数的使用，未定义则关闭 */
#define RT_USING_HOOK

/* 定义了空闲线程的栈大小 */
#define IDLE_THREAD_STACK_SIZE 256

```

(2) 线程间同步与通信部分，该部分会使用到的对象有信号量、互斥量、事件、邮箱、消息队列、信号等。

```

/* 定义该宏可开启信号量的使用，未定义则关闭 */
#define RT_USING_SEMAPHORE

/* 定义该宏可开启互斥量的使用，未定义则关闭 */
#define RT_USING_MUTEX

/* 定义该宏可开启事件集的使用，未定义则关闭 */
#define RT_USING_EVENT

/* 定义该宏可开启邮箱的使用，未定义则关闭 */
#define RT_USING_MAILBOX

/* 定义该宏可开启消息队列的使用，未定义则关闭 */
#define RT_USING_MESSAGEQUEUE

/* 定义该宏可开启信号的使用，未定义则关闭 */
#define RT_USING_SIGNALS

```

(3) 内存管理部分

```

/* 开启静态内存池的使用 */
#define RT_USING_MEMPOOL

/* 定义该宏可开启两个或以上内存堆拼接的使用，未定义则关闭 */
#define RT_USING_MEMHEAP

/* 开启小内存管理算法 */
#define RT_USING_SMALL_MEM

/* 关闭 SLAB 内存管理算法 */
/* #define RT_USING_SLAB */

/* 开启堆的使用 */
#define RT_USING_HEAP

```

(4) 内核设备对象

```
/* 表示开启了系统设备的使用 */
#define RT_USING_DEVICE

/* 定义该宏可开启系统控制台设备的使用，未定义则关闭 */
#define RT_USING_CONSOLE
/* 定义控制台设备的缓冲区大小 */
#define RT_CONSOLEBUF_SIZE 128
/* 控制台设备的名称 */
#define RT_CONSOLE_DEVICE_NAME "uart1"
```

(5) 自动初始化方式

```
/* 定义该宏开启自动初始化机制，未定义则关闭 */
#define RT_USING_COMPONENTS_INIT

/* 定义该宏开启设置应用入口为 main 函数 */
#define RT_USING_USER_MAIN
/* 定义 main 线程的栈大小 */
#define RT_MAIN_THREAD_STACK_SIZE 2048
```

(6) FinSH

```
/* 定义该宏可开启系统 FinSH 调试工具的使用，未定义则关闭 */
#define RT_USING_FINSH

/* 开启系统 FinSH 时：将该线程名称定义为 tshell */
#define FINSH_THREAD_NAME "tshell"

/* 开启系统 FinSH 时：使用历史命令 */
#define FINSH_USING_HISTORY
/* 开启系统 FinSH 时：对历史命令行数的定义 */
#define FINSH_HISTORY_LINES 5

/* 开启系统 FinSH 时：定义该宏开启使用 Tab 键，未定义则关闭 */
#define FINSH_USING_SYMTAB

/* 开启系统 FinSH 时：定义该线程的优先级 */
#define FINSH_THREAD_PRIORITY 20
/* 开启系统 FinSH 时：定义该线程的栈大小 */
#define FINSH_THREAD_STACK_SIZE 4096
/* 开启系统 FinSH 时：定义命令字符长度 */
#define FINSH_CMD_SIZE 80

/* 开启系统 FinSH 时：定义该宏开启 MSH 功能 */
#define FINSH_USING_MSH
/* 开启系统 FinSH 时：开启 MSH 功能时，定义该宏默认使用 MSH 功能 */
#define FINSH_DEFAULT_MSH
```

```
#define FINSH_USING_MSH_DEFAULT
/* 开启系统 FinSH 时：定义该宏，仅使用 MSH 功能 */
#define FINSH_USING_MSH_ONLY
```

(7) 关于 MCU

```
/* 定义该工程使用的 MCU 为 STM32F103ZE；系统通过对芯片类型的定义，来定义芯片的管脚
 */
#define STM32F103ZE

/* 定义时钟源频率 */
#define RT_HSE_VALUE 8000000

/* 定义该宏开启 UART1 的使用 */
#define RT_USING_UART1
```

!!! note “注意事项”在实际应用中，系统配置文件 `rtconfig.h` 是由配置工具自动生成的，无需手动更改。

2.7 常见宏定义说明

RT-Thread 中经常使用一些宏定义，举例 Keil 编译环境下一些常见的宏定义：

1) `rt_inline`，定义如下，`static` 关键字的作用是令函数只能在当前的文件中使用；`inline` 表示内联，用 `static` 修饰后在调用函数时会建议编译器进行内联展开。

```
#define rt_inline           static __inline
```

2) `RT_USED`，定义如下，该宏的作用是向编译器说明这段代码有用，即使函数中没有调用也要保留编译。例如 RT-Thread 自动初始化功能使用了自定义的段，使用 `RT_USED` 会将自定义的代码段保留。

```
#define RT_USED           __attribute__((used))
```

3) `RT_UNUSED`，定义如下，表示函数或变量可能不使用，这个属性可以避免编译器产生警告信息。

```
#define RT_UNUSED          __attribute__((unused))
```

4) `RT_WEAK`，定义如下，常用于定义函数，编译器在链接函数时会优先链接没有该关键字前缀的函数，如果找不到则再链接由 `weak` 修饰的函数。

```
#define RT_WEAK           __weak
```

5) `ALIGN(n)`，定义如下，作用是在给某对象分配地址空间时，将其存放的地址按照 `n` 字节对齐，这里 `n` 可取 2 的幂次方。字节对齐的作用不仅是便于 CPU 快速访问，同时合理的利用字节对齐可以有效地节省存储空间。

```
#define ALIGN(n)           __attribute__((aligned(n)))
```

6) RT_ALIGN(size,align), 定义如下, 作用是将 size 提升为 align 定义的整数的倍数, 例如, RT_ALIGN(13,4) 将返回 16。

```
#define RT_ALIGN(size, align) (((size) + (align) - 1) & ~((align) - 1))
```

第 3 章

线程管理

在日常生活中，我们要完成一个大任务，一般会将它分解成多个简单、容易解决的小问题，小问题逐个被解决，大问题也就随之解决了。在多线程操作系统中，也同样需要开发人员把一个复杂的应用分解成多个小的、可调度的、序列化的程序单元，当合理地划分任务并正确地执行时，这种设计能够让系统满足实时系统的性能及时间的要求，例如让嵌入式系统执行这样的任务，系统通过传感器采集数据，并通过显示屏将数据显示出来，在多线程实时系统中，可以将这个任务分解成两个子任务，如下图所示，一个子任务不间断地读取传感器数据，并将数据写到共享内存中，另外一个子任务周期性的从共享内存中读取数据，并将传感器数据输出到显示屏上。

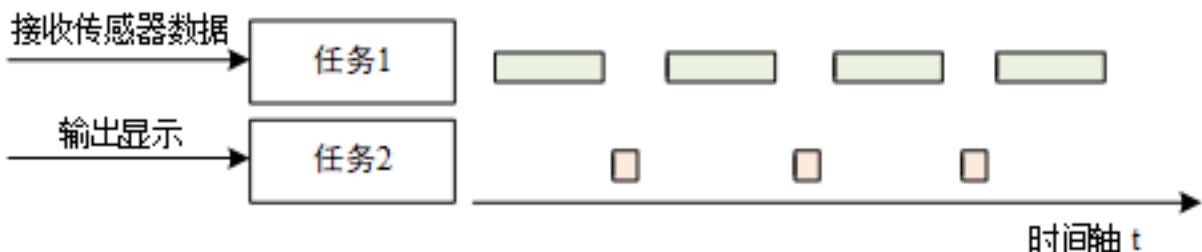


图 3.1：传感器数据接收任务与显示任务的切换执行

在 RT-Thread 中，与上述子任务对应的程序实体就是线程，线程是实现任务的载体，它是 RT-Thread 中最基本的调度单位，它描述了一个任务执行的运行环境，也描述了这个任务所处的优先等级，重要的任务可设置相对较高的优先级，非重要的任务可以设置较低的优先级，不同的任务还可以设置相同的优先级，轮流运行。

当线程运行时，它会认为自己是以独占 CPU 的方式在运行，线程执行时的运行环境称为上下文，具体来说就是各个变量和数据，包括所有的寄存器变量、堆栈、内存信息等。

本章将分成 5 节内容对 RT-Thread 线程管理进行介绍，读完本章，读者会对 RT-Thread 的线程管理机制有比较深入的了解，如：线程有哪些状态、如何创建一个线程、为什么会有空闲线程等问题，心中也会有一个明确的答案了。

3.1 线程管理的功能特点

RT-Thread 线程管理的主要功能是对线程进行管理和调度，系统中总共存在两类线程，分别是系统线程和用户线程，系统线程是由 RT-Thread 内核创建的线程，用户线程是由应用程序创建的线程，这两类线程都会从内核对象容器中分配线程对象，当线程被删除时，也会被从对象容器中删除，如图 3-2 所示，每个线程都有重要的属性，如线程控制块、线程栈、入口函数等。

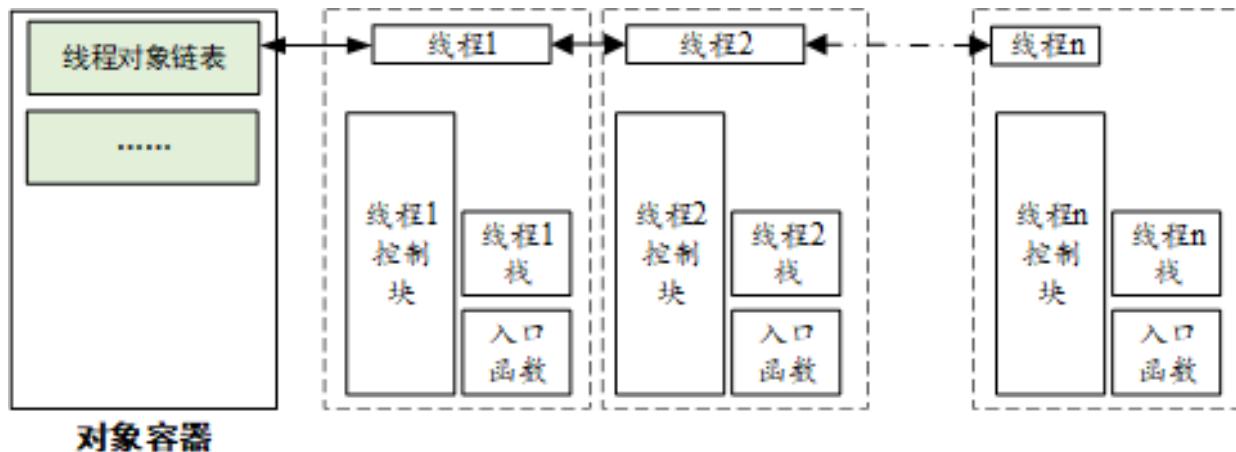


图 3.2: 对象容器与线程对象

RT-Thread 的线程调度器是抢占式的，主要的工作就是从就绪线程列表中查找最高优先级线程，保证最高优先级的线程能够被运行，最高优先级的任务一旦就绪，总能得到 CPU 的使用权。

当一个运行着的线程使一个比它优先级高的线程满足运行条件，当前线程的 CPU 使用权就被剥夺了，或者说被让出了，高优先级的线程立刻得到了 CPU 的使用权。

如果是中断服务程序使一个高优先级的线程满足运行条件，中断完成时，被中断的线程挂起，优先级高的线程开始运行。

当调度器调度线程切换时，先将当前线程上下文保存起来，当再切回到这个线程时，线程调度器将该线程的上下文信息恢复。

3.2 线程的工作机制

3.2.1 线程控制块

在 RT-Thread 中，线程控制块由结构体 struct rt_thread 表示，线程控制块是操作系统用于管理线程的一个数据结构，它会存放线程的一些信息，例如优先级、线程名称、线程状态等，也包含线程与线程之间连接用的链表结构，线程等待事件集合等，详细定义如下：

```
/* 线程控制块 */
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];      /* 线程名称 */
    rt_uint8_t   type;                  /* 对象类型 */
```

```

rt_uint8_t flags;           /* 标志位 */

rt_list_t list;            /* 对象列表 */
rt_list_t tlist;           /* 线程列表 */

/* 栈指针与入口指针 */
void *sp;                  /* 栈指针 */
void *entry;                /* 入口函数指针 */
void *parameter;           /* 参数 */
void *stack_addr;           /* 栈地址指针 */
rt_uint32_t stack_size;     /* 栈大小 */

/* 错误代码 */
rt_err_t error;             /* 线程错误代码 */
rt_uint8_t stat;             /* 线程状态 */

/* 优先级 */
rt_uint8_t current_priority; /* 当前优先级 */
rt_uint8_t init_priority;    /* 初始优先级 */
rt_uint32_t number_mask;

.....
rt_ubase_t init_tick;        /* 线程初始化计数值 */
rt_ubase_t remaining_tick;   /* 线程剩余计数值 */

struct rt_timer thread_timer; /* 内置线程定时器 */

void (*cleanup)(struct rt_thread *tid); /* 线程退出清除函数 */
rt_uint32_t user_data;           /* 用户数据 */
};


```

其中 `init_priority` 是线程创建时指定的线程优先级，在线程运行过程当中是不会被改变的（除非用户执行线程控制函数进行手动调整线程优先级）。`cleanup` 会在线程退出时，被空闲线程回调一次以执行用户设置的清理现场等工作。最后的一个成员 `user_data` 可由用户挂接一些数据信息到线程控制块中，以提供类似线程私有数据的实现。

3.2.2 线程重要属性

3.2.2.1 线程栈

RT-Thread 线程具有独立的栈，当进行线程切换时，会将当前线程的上下文存在栈中，当线程要恢复运行时，再从栈中读取上下文信息，进行恢复。

线程栈还用来存放函数中的局部变量：函数中的局部变量从线程栈空间中申请；函数中局部变量初始时从寄存器中分配（ARM 架构），当这个函数再调用另一个函数时，这些局部变量将放入栈中。

对于线程第一次运行，可以以手工的方式构造这个上下文来设置一些初始的环境：入口函数（PC 寄存器）、入口参数（R0 寄存器）、返回位置（LR 寄存器）、当前机器运行状态（CPSR 寄存器）。

线程栈的增长方向是芯片构架密切相关的，RT-Thread 3.1.0 以前的版本，均只支持栈由高地址向低地址增长的方式，对于 ARM Cortex-M 架构，线程栈可构造如下图所示。



图 3.3: 线程栈 (ARM)

线程栈大小可以这样设定，对于资源相对较大的 MCU，可以适当设计较大的线程栈；也可以在初始时设置较大的栈，例如指定大小为 1K 或 2K 字节，然后在 FinSH 中用 `list_thread` 命令查看线程运行的过程中线程所使用的栈的大小，通过此命令，能够看到从线程启动运行时，到当前时刻点，线程使用的最大栈深度，而后加上适当的余量形成最终的线程栈大小，最后对栈空间大小加以修改。

3.2.2.2 线程状态

线程运行的过程中，同一时间内只允许一个线程在处理器中运行，从运行的过程上划分，线程有多种不同的运行状态，如初始状态、挂起状态、就绪状态等。在 RT-Thread 中，线程包含五种状态，操作系统会自动根据它运行的情况来动态调整它的状态。RT-Thread 中线程的五种状态，如下表所示：

状态	描述
初始状态	当线程刚开始创建还没开始运行时就处于初始状态；在初始状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 <code>RT_THREAD_INIT</code>
就绪状态	在就绪状态下，线程按照优先级排队，等待被执行；一旦当前线程运行完毕让出处理器，操作系统会马上寻找最高优先级的就绪态线程运行。此状态在 RT-Thread 中的宏定义为 <code>RT_THREAD_READY</code>
运行状态	线程当前正在运行。在单核系统中，只有 <code>rt_thread_self()</code> 函数返回的线程处于运行状态；在多核系统中，可能就不止这一个线程处于运行状态。此状态在 RT-Thread 中的宏定义为 <code>RT_THREAD_RUNNING</code>
挂起状态	也称阻塞态。它可能因为资源不可用而挂起等待，或线程主动延时一段时间而挂起。在挂起状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 <code>RT_THREAD_SUSPEND</code>

状态	描述
关闭状态	当线程运行结束时将处于关闭状态。关闭状态的线程不参与线程的调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_CLOSE

3.2.2.3 线程优先级

RT-Thread 线程的优先级是表示线程被调度的优先程度。每个线程都具有优先级，线程越重要，赋予的优先级就应越高，- 线程被调度的可能才会越大。

RT-Thread 最大支持 256 个线程优先级 (0~255)，数值越小的优先级越高，0 为最高优先级。在一些资源比较紧张的系统中，可以根据实际情况选择只支持 8 个或 32 个优先级的系统配置；对于 ARM Cortex-M 系列，普遍采用 32 个优先级。最低优先级默认分配给空闲线程使用，用户一般不使用。在系统中，当有比当前线程优先级更高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理器运行。

3.2.2.4 时间片

每个线程都有时间片这个参数，但时间片仅对优先级相同的就绪态线程有效。系统对优先级相同的就绪态线程采用时间片轮转的调度方式进行调度时，时间片起到约束线程单次运行时长的作用，其单位是一个系统节拍 (OS Tick)，详见第五章。假设有 2 个优先级相同的就绪态线程 A 与 B，A 线程的时间片设置为 10，B 线程的时间片设置为 5，那么当系统中不存在比 A 优先级高的就绪态线程时，系统会在 A、B 线程间来回切换执行，并且每次对 A 线程执行 10 个节拍的时长，对 B 线程执行 5 个节拍的时长，如下图。

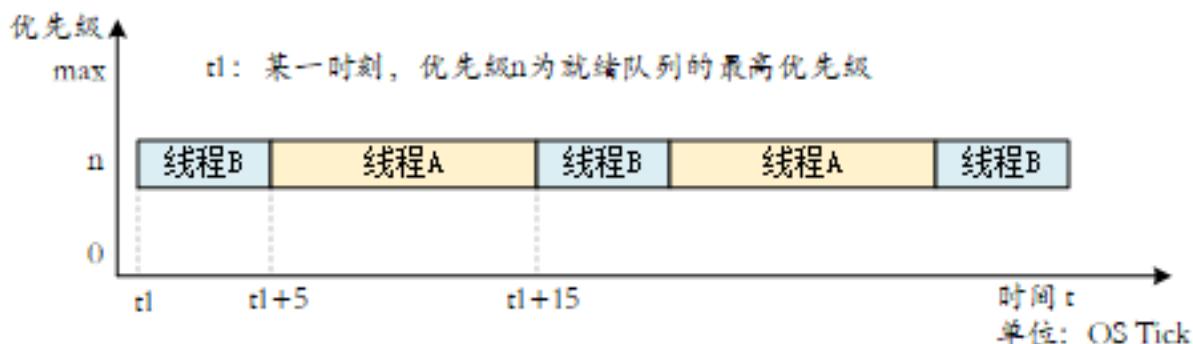


图 3.4: 相同优先级时间片轮转

3.2.2.5 线程的入口函数

线程控制块中的 `entry` 是线程的入口函数，它是线程实现预期功能的函数。线程的入口函数由用户设计实现，一般有以下两种代码形式：

-无限循环模式：

在实时系统中，线程通常是被动式的：这个是由实时系统的特性所决定的，实时系统通常总是等待外界事件的发生，而后进行相应的服务：

```
void thread_entry(void* parameter)
{
    while (1)
```

```
{
/* 等待事件的发生 */

/* 对事件进行服务、进行处理 */
}

}
```

线程看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是作为一个实时系统，一个优先级明确的实时系统，如果一个线程中的程序陷入了死循环操作，那么比它优先级低的线程都将不能够得到执行。所以在实时操作系统中必须注意的一点就是：线程中不能陷入死循环操作，必须要有让出 CPU 使用权的动作，如循环中调用延时函数或者主动挂起。用户设计这种无线循环的线程的目的，就是为了让这个线程一直被系统循环调度运行，永不删除。

-顺序执行或有限次循环模式：

如简单的顺序语句、`do whlie()` 或 `for()` 循环等，此类线程不会循环或不会永久循环，可谓是“一次性”线程，一定会被执行完毕。在执行完毕后，线程将被系统自动删除。

```
static void thread_entry(void* parameter)
{
    /* 处理事务 #1 */
    ...
    /* 处理事务 #2 */
    ...
    /* 处理事务 #3 */
}
```

3.2.2.6 线程错误码

一个线程就是一个执行场景，错误码是与执行环境密切相关的，所以每个线程配备了一个变量用于保存错误码，线程的错误码有以下几种：

<code>#define RT_EOK</code>	0 /* 无错误 */
<code>#define RT_ERROR</code>	1 /* 普通错误 */
<code>#define RTETIMEOUT</code>	2 /* 超时错误 */
<code>#define RT_EFULL</code>	3 /* 资源已满 */
<code>#define RT_EEMPTY</code>	4 /* 无资源 */
<code>#define RT_ENOMEM</code>	5 /* 无内存 */
<code>#define RT_ENOSYS</code>	6 /* 系统不支持 */
<code>#define RT_EBUSY</code>	7 /* 系统忙 */
<code>#define RT_EIO</code>	8 /* IO 错误 */
<code>#define RT_EINTR</code>	9 /* 中断系统调用 */
<code>#define RT_EINVAL</code>	10 /* 非法参数 */

3.2.3 线程状态切换

RT-Thread 提供一系列的操作系统调用接口，使得线程的状态在这五个状态之间来回切换。几种状态间的转换关系如下图所示：

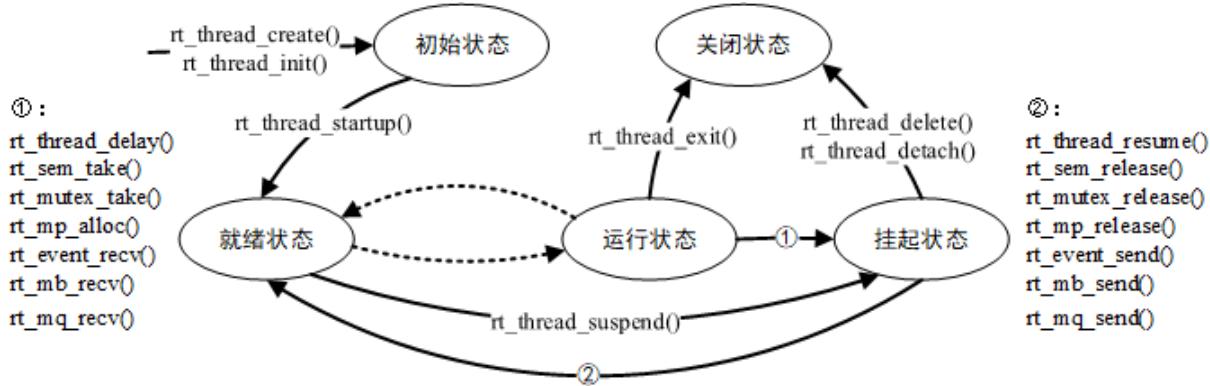


图 3.5: 线程状态转换图

线程通过调用函数 `rt_thread_create/init()` 进入到初始状态（`RT_THREAD_INIT`）；初始状态的线程通过调用函数 `rt_thread_startup()` 进入到就绪状态（`RT_THREAD_READY`）；就绪状态的线程被调度器调度后进入运行状态（`RT_THREAD_RUNNING`）；当处于运行状态的线程调用 `rt_thread_delay()`, `rt_sem_take()`, `rt_mutex_take()`, `rt_mb_recv()` 等函数或者获取不到资源时，将进入到挂起状态（`RT_THREAD_SUSPEND`）；处于挂起状态的线程，如果等待超时依然未能获得资源或由于其他线程释放了资源，那么它将返回到就绪状态。挂起状态的线程，如果调用 `rt_thread_delete/detach()` 函数，将更改为关闭状态（`RT_THREAD_CLOSE`）；而运行状态的线程，如果运行结束，就会在线程的最后部分执行 `rt_thread_exit()` 函数，将状态更改为关闭状态。

!!! note “注意事项” RT-Thread 中，实际上线程并不存在运行状态，就绪状态和运行状态是等同的。

3.2.4 系统线程

前文中已提到，系统线程是指由系统创建的线程，用户线程是由用户程序调用线程管理接口创建的线程，在 RT-Thread 内核中的系统线程有空闲线程和主线程。

3.2.4.1 空闲线程

空闲线程是系统创建的最低优先级的线程，线程状态永远为就绪态。当系统中无其他就绪线程存在时，调度器将调度到空闲线程，它通常是一个死循环，且永远不能被挂起。另外，空闲线程在 RT-Thread 也有着它的特殊用途：

若某线程运行完毕，系统将自动删除线程：自动执行 `rt_thread_exit()` 函数，先将该线程从系统就绪队列中删除，再将该线程的状态更改为关闭状态，不再参与系统调度，然后挂入 `rt_thread_defunct` 僵尸队列（资源未回收、处于关闭状态的线程队列）中，最后空闲线程会回收被删除线程的资源。

空闲线程也提供了接口来运行用户设置的钩子函数，在空闲线程运行时会调用该钩子函数，适合钩入功耗管理、看门狗喂狗等工作。

3.2.4.2 主线程

在系统启动时，系统会创建 `main` 线程，它的入口函数为 `main_thread_entry0`，用户的应用入口函数 `main()` 就是从这里真正开始的，系统调度器启动后，`main` 线程就开始运行，过程如下图，用户可以在 `main()` 函数里添加自己的应用程序初始化代码。

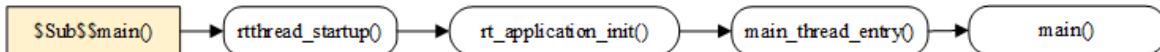


图 3.6: 主线程调用过程

3.3 线程的管理方式

本章前面 2 节对线程的功能与工作机制进行了概念上的讲解，相信大家对线程已经不再陌生。本节将深入到 RT-Thread 线程的各个接口，并给出部分源码，帮助读者在代码层次上理解线程。

下图描述了线程的相关操作，包含：创建 / 初始化线程、启动线程、运行线程、删除 / 脱离线程。可以使用 `rt_thread_create()` 创建一个动态线程，使用 `rt_thread_init()` 初始化一个静态线程，动态线程与静态线程的区别是：动态线程是系统自动从动态内存堆上分配栈空间与线程句柄（初始化 `heap` 之后才能使用 `create` 创建动态线程），静态线程是由用户分配栈空间与线程句柄。

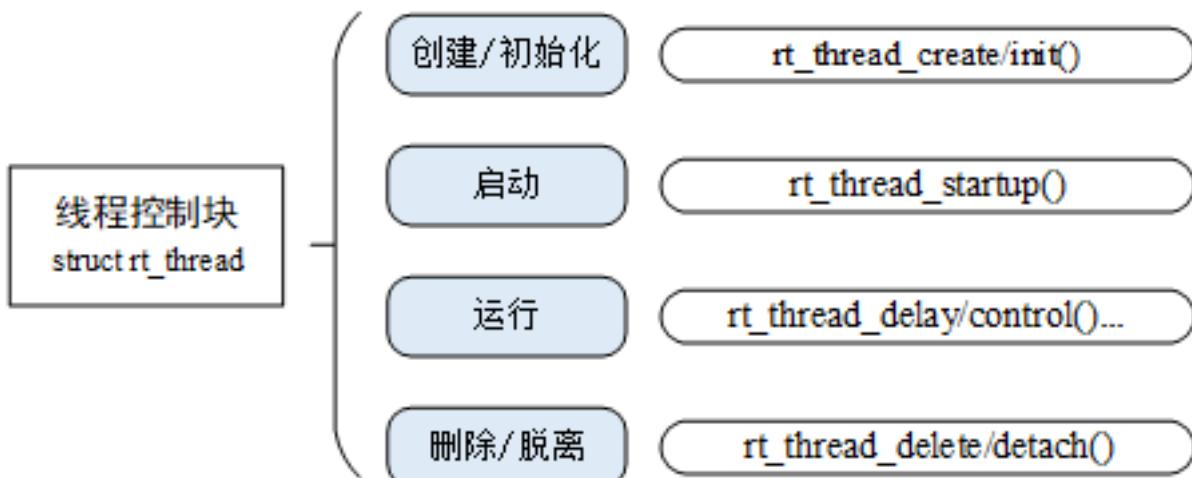


图 3.7: 线程相关操作

3.3.1 创建和删除线程

一个线程要成为可执行的对象，就必须由操作系统的内核来为它创建一个线程。可以通过如下的接口创建一个动态线程：

```

rt_thread_t rt_thread_create(const char* name,
                             void (*entry)(void* parameter),
                             void* parameter,
                             rt_uint32_t stack_size,
                             rt_uint8_t priority,
                             rt_uint32_t tick);

```

调用这个函数时，系统会从动态堆内存中分配一个线程句柄以及按照参数中指定的栈大小从动态堆内存中分配相应的空间。分配出来的栈空间是按照 `rtconfig.h` 中配置的 `RT_ALIGN_SIZE` 方式对齐。线程创建 `rt_thread_create()` 的参数和返回值见下表：

参数	描述
name	线程的名称；线程名称的最大长度由 <code>rtconfig.h</code> 中的宏 <code>RT_NAME_MAX</code> 指定，多余部分会被自动截掉
entry	线程入口函数
parameter	线程入口函数参数
stack_size	线程栈大小，单位是字节
priority	线程的优先级。优先级范围根据系统配置情况 (<code>rtconfig.h</code> 中的 <code>RT_THREAD_PRIORITY_MAX</code> 宏定义)，如果支持的是 256 级优先级，那么范围是从 0~255，数值越小优先级越高，0 代表最高优先级
tick	线程的时间片大小。时间片 (tick) 的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束后，调度器自动选择下一个就绪态的同优先级线程进行运行
返回	—
thread	线程创建成功，返回线程句柄
<code>RT_NULL</code>	线程创建失败

对于一些使用 `rt_thread_create()` 创建出来的线程，当不需要使用，或者运行出错时，我们可以使用下面的函数接口来从系统中把线程完全删除掉：

```
rt_err_t rt_thread_delete(rt_thread_t thread);
```

调用该函数后，线程对象将会被移出线程队列并且从内核对象管理器中删除，线程占用的堆栈空间也会被释放，收回的空间将重新用于其他的内存分配。实际上，用 `rt_thread_delete()` 函数删除线程接口，仅仅是把相应的线程状态更改为 `RT_THREAD_CLOSE` 状态，然后放入到 `rt_thread_defunct` 队列中；而真正的删除动作（释放线程控制块和释放线程栈）需要到下一次执行空闲线程时，由空闲线程完成最后的线程删除动作。线程删除 `rt_thread_delete()` 接口的参数和返回值见下表：

参数	描述
thread	要删除的线程句柄
返回	—
<code>RT_EOK</code>	删除线程成功
<code>-RT_ERROR</code>	删除线程失败

这个函数仅在使能了系统动态堆时才有效（即 `RT_USING_HEAP` 宏定义已经定义了）。

3.3.2 初始化和脱离线程

线程的初始化可以使用下面的函数接口完成，来初始化静态线程对象：

```
rt_err_t rt_thread_init(struct rt_thread* thread,
```

```
const char* name,
void (*entry)(void* parameter), void* parameter,
void* stack_start, rt_uint32_t stack_size,
rt_uint8_t priority, rt_uint32_t tick);
```

静态线程的线程句柄（或者说线程控制块指针）、线程栈由用户提供。静态线程是指线程控制块、线程运行栈一般都设置为全局变量，在编译时就被确定、被分配处理，内核不负责动态分配内存空间。需要注意的是，用户提供的栈首地址需做系统对齐（例如 ARM 上需要做 4 字节对齐）。线程初始化接口 `rt_thread_init()` 的参数和返回值见下表：

参数	描述
<code>thread</code>	线程句柄。线程句柄由用户提供出来，并指向对应的线程控制块内存地址
<code>name</code>	线程的名称；线程名称的最大长度由 <code>rtconfig.h</code> 中定义的 <code>RT_NAME_MAX</code> 宏指定，多余部分会被自动截掉
<code>entry</code>	线程入口函数
<code>parameter</code>	线程入口函数参数
<code>stack_start</code>	线程栈起始地址
<code>stack_size</code>	线程栈大小，单位是字节。在大多数系统中需要做栈空间地址对齐（例如 ARM 体系结构中需要向 4 字节地址对齐）
<code>priority</code>	线程的优先级。优先级范围根据系统配置情况（ <code>rtconfig.h</code> 中的 <code>RT_THREAD_PRIORITY_MAX</code> 宏定义），如果支持的是 256 级优先级，那么范围是从 0 ~ 255，数值越小优先级越高，0 代表最高优先级
<code>tick</code>	线程的时间片大小。时间片（ <code>tick</code> ）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行
返回	—
<code>RT_EOK</code>	线程创建成功
<code>-RT_ERROR</code>	线程创建失败

对于用 `rt_thread_init()` 初始化的线程，使用 `rt_thread_detach()` 将使线程对象在线程队列和内核对象管理器中被脱离。线程脱离函数如下：

```
rt_err_t rt_thread_detach (rt_thread_t thread);
```

线程脱离接口 `rt_thread_detach()` 的参数和返回值见下表：

参数	描述
<code>thread</code>	线程句柄，它应该是由 <code>rt_thread_init</code> 进行初始化的线程句柄。
返回	—
<code>RT_EOK</code>	线程脱离成功
<code>-RT_ERROR</code>	线程脱离失败

参数	描述
----	----

这个函数接口是和 `rt_thread_delete()` 函数相对应的，`rt_thread_delete()` 函数操作的对象是 `rt_thread_create()` 创建的句柄，而 `rt_thread_detach()` 函数操作的对象是使用 `rt_thread_init()` 函数初始化的线程控制块。同样，线程本身不应调用这个接口脱离线程本身。

3.3.3 启动线程

创建（初始化）的线程状态处于初始状态，并未进入就绪线程的调度队列，我们可以在线程初始化 / 创建成功后调用下面的函数接口让该线程进入就绪态：

```
rt_err_t rt_thread_startup(rt_thread_t thread);
```

当调用这个函数时，将把线程的状态更改为就绪状态，并放到相应优先级队列中等待调度。如果新启动的线程优先级比当前线程优先级高，将立刻切换到这个线程。线程启动接口 `rt_thread_startup()` 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	—
RT_EOK	线程启动成功
-RT_ERROR	线程起动失败

3.3.4 获得当前线程

在程序的运行过程中，相同的一段代码可能会被多个线程执行，在执行的时候可以通过下面的函数接口获得当前执行的线程句柄：

```
rt_thread_t rt_thread_self(void);
```

该接口的返回值见下表：

返回	描述
thread	当前运行的线程句柄
RT_NULL	失败，调度器还未启动

3.3.5 使线程让出处理器资源

当前线程的时间片用完或者该线程主动要求让出处理器资源时，它将不再占有处理器，调度器会选择相同优先级的下一个线程执行。线程调用这个接口后，这个线程仍然在就绪队列中。线程让出处理器使用

下面的函数接口：

```
rt_err_t rt_thread_yield(void);
```

调用该函数后，当前线程首先把自己从它所在的就绪优先级线程队列中删除，然后把自己挂到这个优先级队列链表的尾部，然后激活调度器进行线程上下文切换（如果当前优先级只有这一个线程，则这个线程继续执行，不进行上下文切换动作）。

`rt_thread_yield()` 函数和 `rt_schedule()` 函数比较相像，但在有相同优先级的其他就绪态线程存在时，系统的行为却完全不一样。执行 `rt_thread_yield()` 函数后，当前线程被换出，相同优先级的下一个就绪线程将被执行。而执行 `rt_schedule()` 函数后，当前线程并不一定被换出，即使被换出，也不会被放到就绪线程链表的尾部，而是在系统中选取就绪的优先级最高的线程执行（如果系统中没有比当前线程优先级更高的线程存在，那么执行完 `rt_schedule()` 函数后，系统将继续执行当前线程）。

3.3.6 使线程睡眠

在实际应用中，我们有时需要让运行的当前线程延迟一段时间，在指定的时间到达后重新运行，这就叫做“线程睡眠”。线程睡眠可使用以下三个函数接口：

```
rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);
rt_err_t rt_thread_mdelay(rt_int32_t ms);
```

这三个函数接口的作用相同，调用它们可以使当前线程挂起一段指定的时间，当这个时间过后，线程会被唤醒并再次进入就绪状态。这个函数接受一个参数，该参数指定了线程的休眠时间。线程睡眠接口 `rt_thread_sleep/delay/mdielay()` 的参数和返回值见下表：

参数	描述
tick/ms	线程睡眠的时间：sleep/delay 的传入参数 tick 以 1 个 OS Tick 为单位；mdielay 的传入参数 ms 以 1ms 为单位；
返回	—
RT_EOK	操作成功

3.3.7 挂起和恢复线程

当线程调用 `rt_thread_delay()` 时，线程将主动挂起；当调用 `rt_sem_take()`，`rt_mb_recv()` 等函数时，资源不可使用也将导致线程挂起。处于挂起状态的线程，如果其等待的资源超时（超过其设定的等待时间），那么该线程将不再等待这些资源，并返回到就绪状态；或者，当其他线程释放掉该线程所等待的资源时，该线程也会返回到就绪状态。

线程挂起使用下面的函数接口：

```
rt_err_t rt_thread_suspend (rt_thread_t thread);
```

线程挂起接口 `rt_thread_suspend()` 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	—
RT_EOK	线程挂起成功
-RT_ERROR	线程挂起失败，因为该线程的状态并不是就绪状态

!!! note “注意事项”通常不应该使用这个函数来挂起线程本身，如果确实需要采用 `rt_thread_suspend()` 函数挂起当前任务，需要在调用 `rt_thread_suspend()` 函数后立刻调用 `rt_schedule()` 函数进行手动的线程上下文切换。用户只需要了解该接口的作用，不推荐使用该接口。

恢复线程就是让挂起的线程重新进入就绪状态，并将线程放入系统的就绪队列中；如果被恢复线程在所有就绪态线程中，位于最高优先级链表的第一位，那么系统将进行线程上下文的切换。线程恢复使用下面的函数接口：

```
rt_err_t rt_thread_resume(rt_thread_t thread);
```

线程恢复接口 `rt_thread_resume()` 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	—
RT_EOK	线程恢复成功
-RT_ERROR	线程恢复失败，因为该线程的状态并不是 RT_THREAD_SUSPEND 状态

3.3.8 控制线程

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下函数接口：

```
rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg);
```

线程控制接口 `rt_thread_control()` 的参数和返回值见下表：

函数参数	描述
thread	线程句柄
cmd	指示控制命令
arg	控制参数
返回	—
RT_EOK	控制执行正确
-RT_ERROR	失败

指示控制命令 cmd 当前支持的命令包括：

- RT_THREAD_CTRL_CHANGE_PRIORITY:** 动态更改线程的优先级；
- RT_THREAD_CTRL_STARTUP:** 开始运行一个线程，等同于 `rt_thread_startup()` 函数调用；
- RT_THREAD_CTRL_CLOSE:** 关闭一个线程，等同于 `rt_thread_delete()` 函数调用。

3.3.9 设置和删除空闲钩子

空闲钩子函数是空闲线程的钩子函数，如果设置了空闲钩子函数，就可以在系统执行空闲线程时，自动执行空闲钩子函数来做一些其他事情，比如系统指示灯。设置 / 删除空闲钩子的接口如下：

```
rt_err_t rt_thread_idle_sethook(void (*hook)(void));
rt_err_t rt_thread_idle_delhook(void (*hook)(void));
```

设置空闲钩子函数 `rt_thread_idle_sethook()` 的输入参数和返回值如下表所示：

函数参数	描述
hook	设置的钩子函数
返回	——
RT_EOK	设置成功
-RT_EFULL	设置失败

删除空闲钩子函数 `rt_thread_idle_delhook()` 的输入参数和返回值如下表所示：

函数参数	描述
hook	删除的钩子函数
返回	——
RT_EOK	删除成功
-RT_ENOSYS	删除失败

!!! note “注意事项” 空闲线程是一个线程状态永远为就绪态的线程，因此设置的钩子函数必须保证空闲线程在任何时刻都不会处于挂起状态，例如 `rt_thread_delay()`, `rt_sem_take()` 等可能会导致线程挂起的函数都不能使用。

3.3.10 设置调度器钩子

在整个系统的运行时，系统都处于线程运行、中断触发 - 响应中断、切换到其他线程，甚至是线程间的切换过程中，或者说系统的上下文切换是系统中最普遍的事件。有时用户可能会想知道在一个时刻发生了什么样的线程切换，可以通过调用下面的函数接口设置一个相应的钩子函数。在系统线程切换时，这个钩子函数将被调用：

```
void rt_scheduler_sethook(void (*hook)(struct rt_thread* from, struct rt_thread* to));
);
```

设置调度器钩子函数的输入参数如下表所示：

函数参数	描述
hook	表示用户定义的钩子函数指针

钩子函数 hook() 的声明如下：

```
void hook(struct rt_thread* from, struct rt_thread* to);
```

调度器钩子函数 hook() 的输入参数如下表所示：

函数参数	描述
from	表示系统所要切换出的线程控制块指针
to	表示系统所要切换到的线程控制块指针

!!! note “注意事项” 请仔细编写你的钩子函数，稍有不慎将很可能导致整个系统运行不正常（在这个钩子函数中，基本上不允许调用系统 API，更不应该导致当前运行的上下文挂起）。

3.4 线程应用示例

下面给出在 Keil 模拟器环境下的应用示例。

3.4.1 创建线程示例

这个例子创建一个动态线程初始化一个静态线程，一个线程在运行完毕后自动被系统删除，另一个线程一直打印计数，如下代码：

```
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

static rt_thread_t tid1 = RT_NULL;

/* 线程 1 的入口函数 */
static void thread1_entry(void *parameter)
{
    rt_uint32_t count = 0;
```

```
while (1)
{
    /* 线程 1 采用低优先级运行，一直打印计数值 */
    rt_kprintf("thread1 count: %d\n", count++);
    rt_thread_mdelay(500);
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
/* 线程 2 入口 */
static void thread2_entry(void *param)
{
    rt_uint32_t count = 0;

    /* 线程 2 拥有较高的优先级，以抢占线程 1 而获得执行 */
    for (count = 0; count < 10 ; count++)
    {
        /* 线程 2 打印计数值 */
        rt_kprintf("thread2 count: %d\n", count);
    }
    rt_kprintf("thread2 exit\n");
    /* 线程 2 运行结束后也将自动被系统脱离 */
}

/* 线程示例 */
int thread_sample(void)
{
    /* 创建线程 1，名称是 thread1，入口是 thread1_entry*/
    tid1 = rt_thread_create("thread1",
                           thread1_entry, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);

    /* 如果获得线程控制块，启动这个线程 */
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 初始化线程 2，名称是 thread2，入口是 thread2_entry */
    rt_thread_init(&thread2,
                  "thread2",
                  thread2_entry,
                  RT_NULL,
                  &thread2_stack[0],
                  sizeof(thread2_stack),
                  THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);
```

```

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(thread_sample, thread sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \    3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >thread_sample
msh >thread2 count: 0
thread2 count: 1
thread2 count: 2
thread2 count: 3
thread2 count: 4
thread2 count: 5
thread2 count: 6
thread2 count: 7
thread2 count: 8
thread2 count: 9
thread2 exit
thread1 count: 0
thread1 count: 1
thread1 count: 2
thread1 count: 3
...

```

线程 2 计数到一定值会执行完毕，线程 2 被系统自动删除，计数停止。线程 1 一直打印计数。

!!! note “注意事项” 关于删除线程：大多数线程是循环执行的，无需删除；而能运行完毕的线程，RT-Thread 在线程运行完毕后，自动删除线程，在 `rt_thread_exit()` 里完成删除动作。用户只需要了解该接口的作用，不推荐使用该接口（可以由其他线程调用此接口或在定时器超时函数中调用此接口删除一个线程，但是这种使用非常少）。

3.4.2 线程时间片轮转调度示例

这个例子创建两个线程，在执行时会一直打印计数，如下代码：

```

#include <rtthread.h>

#define THREAD_STACK_SIZE    1024
#define THREAD_PRIORITY       20
#define THREAD_TIMESLICE     10

/* 线程入口 */

```

```
static void thread_entry(void* parameter)
{
    rt_uint32_t value;
    rt_uint32_t count = 0;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        if(0 == (count % 5))
        {
            rt_kprintf("thread %d is running ,thread %d count = %d\n", value , value
                       , count);

            if(count> 200)
                return;
        }
        count++;
    }
}

int timeslice_sample(void)
{
    rt_thread_t tid = RT_NULL;
    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                          thread_entry, (void*)1,
                          THREAD_STACK_SIZE,
                          THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid != RT_NULL)
        rt_thread_startup(tid);

    /* 创建线程 2 */
    tid = rt_thread_create("thread2",
                          thread_entry, (void*)2,
                          THREAD_STACK_SIZE,
                          THREAD_PRIORITY, THREAD_TIMESLICE-5);
    if (tid != RT_NULL)
        rt_thread_startup(tid);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(timeslice_sample, timeslice sample);
```

仿真运行结果如下：

```
\ | /
- RT -      Thread Operating System
```

```

/ | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >timeslice_sample
msh >thread 1 is running ,thread 1 count = 0
thread 1 is running ,thread 1 count = 5
thread 1 is running ,thread 1 count = 10
thread 1 is running ,thread 1 count = 15
...
thread 1 is running ,thread 1 count = 125
thread 1 is rthread 2 is running ,thread 2 count = 0
thread 2 is running ,thread 2 count = 5
thread 2 is running ,thread 2 count = 10
thread 2 is running ,thread 2 count = 15
thread 2 is running ,thread 2 count = 20
thread 2 is running ,thread 2 count = 25
thread 2 is running ,thread 2 count = 30
thread 2 is running ,thread 2 count = 35
thread 2 is running ,thread 2 count = 40
thread 2 is running ,thread 2 count = 45
thread 2 is running ,thread 2 count = 50
thread 2 is running ,thread 2 count = 55
thread 2 is running ,thread 2 count = 60
thread 2 is running ,thread 2 count = 65
thread 1 is running ,thread 1 count = 135
...
thread 2 is running ,thread 2 count = 205

```

由运行的计数结果可以看出，线程 2 的运行时间是线程 1 的一半。

3.4.3 线程调度器钩子示例

在线程进行调度切换时，会执行调度，我们可以设置一个调度器钩子，这样可以在线程切换时，做一些额外的事情，这个例子是在调度器钩子函数中打印线程间的切换信息，如下代码：

```

#include <rtthread.h>

#define THREAD_STACK_SIZE    1024
#define THREAD_PRIORITY      20
#define THREAD_TIMESLICE     10

/* 针对每个线程的计数器 */
volatile rt_uint32_t count[2];

/* 线程 1、2 共用一个入口，但入口参数不同 */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;

    value = (rt_uint32_t)parameter;

```

```

while (1)
{
    rt_kprintf("thread %d is running\n", value);
    rt_thread_mdelay(1000); // 延时一段时间
}
}

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

static void hook_of_scheduler(struct rt_thread* from, struct rt_thread* to)
{
    rt_kprintf("from: %s --> to: %s \n", from->name , to->name);
}

int scheduler_hook(void)
{
    /* 设置调度器钩子 */
    rt_scheduler_sethook(hook_of_scheduler);

    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                           thread_entry, (void*)1,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 创建线程 2 */
    tid2 = rt_thread_create("thread2",
                           thread_entry, (void*)2,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE - 5);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(scheduler_hook, scheduler_hook sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh > scheduler_hook
msh >from: tshell --> to: thread1

```

```
thread 1 is running
from: thread1 --> to: thread2
thread 2 is running
from: thread2 --> to: tidle
from: tidle --> to: thread1
thread 1 is running
from: thread1 --> to: tidle
from: tidle --> to: thread2
thread 2 is running
from: thread2 --> to: tidle
...
...
```

由仿真的结果可以看出，对线程进行切换时，设置的调度器钩子函数是在正常工作的，一直在打印线程切换的信息，包含切换到空闲线程。

第 4 章

时钟管理

时间是非常重要的概念，和朋友出去游玩需要约定时间，完成任务也需要花费时间，生活离不开时间。操作系统也一样，需要通过时间来规范其任务的执行，操作系统中最小的时间单位是时钟节拍 (OS Tick)。本章主要介绍时钟节拍和基于时钟节拍的定时器，读完本章，我们将了解时钟节拍如何产生，并学会如何使用 RT-Thread 的定时器。

4.1 时钟节拍

任何操作系统都需要提供一个时钟节拍，以供系统处理所有和时间有关的事件，如线程的延时、线程的时间片轮转调度以及定时器超时等。时钟节拍是特定的周期性中断，这个中断可以看做是系统心跳，中断之间的时间间隔取决于不同的应用，一般是 1ms-100ms，时钟节拍率越快，系统的额外开销就越大，从系统启动开始计数的时钟节拍数称为系统时间。

RT-Thread 中，时钟节拍的长度可以根据 RT_TICK_PER_SECOND 的定义来调整，等于 1/RT_TICK_PER_SECOND 秒。

4.1.1 时钟节拍的实现方式

时钟节拍由配置为中断触发模式的硬件定时器产生，当中断到来时，将调用一次：void rt_tick_increase(void)，通知操作系统已经过去一个系统时钟；不同硬件定时器中断实现都不同，下面的中断函数以 STM32 定时器作为示例。

```
void SysTick_Handler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();
    .....
    rt_tick_increase();
    /* 退出中断 */
    rt_interrupt_leave();
}
```

在中断函数中调用 rt_tick_increase() 对全局变量 rt_tick 进行自加，代码如下所示：

```

void rt_tick_increase(void)
{
    struct rt_thread *thread;

    /* 全局变量 rt_tick 自加 */
    ++ rt_tick;

    /* 检查时间片 */
    thread = rt_thread_self();

    -- thread->remaining_tick;
    if (thread->remaining_tick == 0)
    {
        /* 重新赋初值 */
        thread->remaining_tick = thread->init_tick;

        /* 线程挂起 */
        rt_thread_yield();
    }

    /* 检查定时器 */
    rt_timer_check();
}

```

可以看到全局变量 `rt_tick` 在每经过一个时钟节拍时，值就会加 1，`rt_tick` 的值表示了系统从启动开始总共经过的时钟节拍数，即系统时间。此外，每经过一个时钟节拍时，都会检查当前线程的时间片是否用完，以及是否有定时器超时。

!!! note “注意事项” 中断中的 `rt_timer_check()` 用于检查系统硬件定时器链表，如果有定时器超时，将调用相应的超时函数。且所有定时器在定时超时后都会从定时器链表中被移除，而周期性定时器会在它再次启动时被加入定时器链表。

4.1.2 获取时钟节拍

由于全局变量 `rt_tick` 在每经过一个时钟节拍时，值就会加 1，通过调用 `rt_tick_get` 会返回当前 `rt_tick` 的值，即可以获取到当前的时钟节拍值。此接口可用于记录系统的运行时间长短，或者测量某任务运行的时间。接口函数如下：

```
rt_tick_t rt_tick_get(void);
```

下表描述了 `rt_tick_get()` 函数的返回值：

返回	描述
<code>rt_tick</code>	当前时钟节拍值

4.2 定时器管理

定时器，是指从指定的时刻开始，经过一定的指定时间后触发一个事件，例如定个时间提醒第二天能够按时起床。定时器有硬件定时器和软件定时器之分：

1) 硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。

2) 软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。

RT-Thread 操作系统提供软件实现的定时器，以时钟节拍（OS Tick）的时间长度为单位，即定时数值必须是 OS Tick 的整数倍，例如一个 OS Tick 是 10ms，那么上层软件定时器只能是 10ms, 20ms, 100ms 等，而不能定时为 15ms。RT-Thread 的定时器也基于系统的节拍，提供了基于节拍整数倍的定时能力。

4.2.1 RT-Thread 定时器介绍

RT-Thread 的定时器提供两类定时器机制：第一类是单次触发定时器，这类定时器在启动后只会触发一次定时器事件，然后定时器自动停止。第二类是周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动的停止，否则将永远持续执行下去。

另外，根据超时函数执行时所处的上下文环境，RT-Thread 的定时器可以分为 HARD_TIMER 模式与 SOFT_TIMER 模式，如下图。

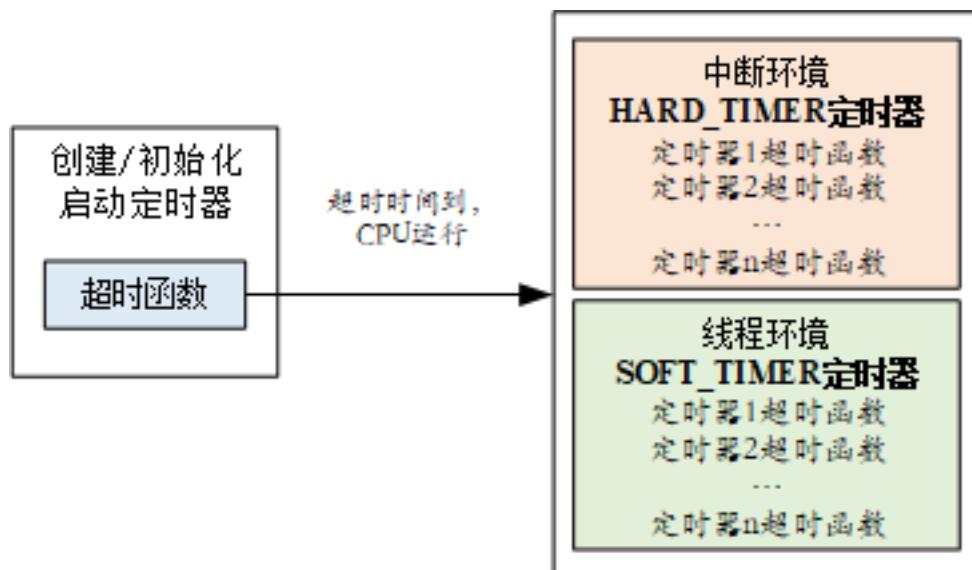


图 4.1: 定时器上下文环境

4.2.1.1 HARD_TIMER 模式

HARD_TIMER 模式的定时器超时函数在中断上下文环境中执行，可以在初始化 / 创建定时器时使用参数 RT_TIMER_FLAG_HARD_TIMER 来指定。

在中断上下文环境中执行时，对于超时函数的要求与中断服务例程的要求相同：执行时间应该尽量短，执行时不应导致当前上下文挂起、等待。例如在中断上下文中执行的超时函数它不应该试图去申请动态内存、释放动态内存等。

RT-Thread 定时器默认的方式是 HARD_TIMER 模式，即定时器超时后，超时函数是在系统时钟中断的上下文环境中运行的。在中断上下文中的执行方式决定了定时器的超时函数不应该调用任何会让当前上下文挂起的系统函数；也不能够执行非常长的时间，否则会导致其他中断的响应时间加长或抢占了其他线程执行的时间。

4.2.1.2 SOFT_TIMER 模式

SOFT_TIMER 模式可配置，通过宏定义 RT_USING_TIMER_SOFT 来决定是否启用该模式。该模式被启用后，系统会在初始化时创建一个 timer 线程，然后 SOFT_TIMER 模式的定时器超时函数在都会在 timer 线程的上下文环境中执行。可以在初始化 / 创建定时器时使用参数 RT_TIMER_FLAG_SOFT_TIMER 来指定设置 SOFT_TIMER 模式。

4.2.2 定时器工作机制

下面以一个例子来说明 RT-Thread 定时器的工作机制。在 RT-Thread 定时器模块中维护着两个重要的全局变量：

- (1) 当前系统经过的 tick 时间 rt_tick（当硬件定时器中断来临时，它将加 1）；
- (2) 定时器链表 rt_timer_list。系统新创建并激活的定时器都会按照以超时时间排序的方式插入到 rt_timer_list 链表中。

如下图所示，系统当前 tick 值为 20，在当前系统中已经创建并启动了三个定时器，分别是定时时间为 50 个 tick 的 Timer1、100 个 tick 的 Timer2 和 500 个 tick 的 Timer3，这三个定时器分别加上系统当前时间 rt_tick=20，从小到大排序链接在 rt_timer_list 链表中，形成如图所示的定时器链表结构。

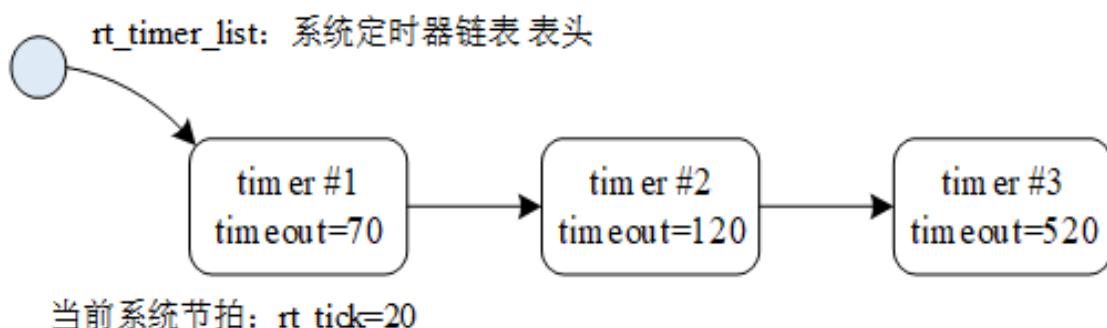


图 4.2: 定时器链表示意图

而 rt_tick 随着硬件定时器的触发一直在增长（每一次硬件定时器中断来临，rt_tick 变量会加 1），50 个 tick 以后，rt_tick 从 20 增长到 70，与 Timer1 的 timeout 值相等，这时会触发与 Timer1 定时器相关联的超时函数，同时将 Timer1 从 rt_timer_list 链表上删除。同理，100 个 tick 和 500 个 tick 过去后，与 Timer2 和 Timer3 定时器相关联的超时函数会被触发，接着将 Timer2 和 Timer3 定时器从 rt_timer_list 链表中删除。

如果系统当前定时器状态在 10 个 tick 以后 ($rt_tick=30$) 有一个任务新创建了一个 tick 值为 300 的 Timer4 定时器, 由于 Timer4 定时器的 $timeout=rt_tick+300=330$, 因此它将被插入到 Timer2 和 Timer3 定时器中间, 形成如下图所示链表结构:

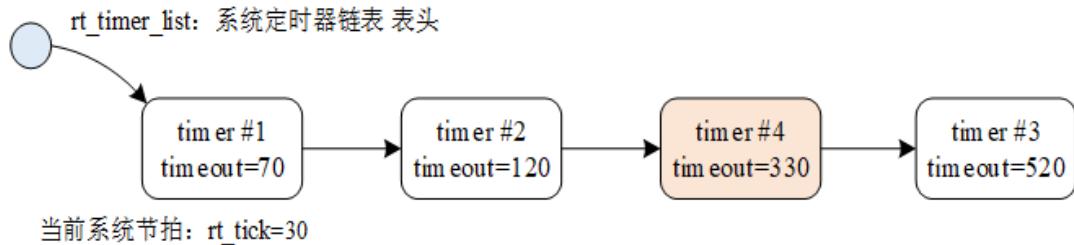


图 4.3: 定时器链表插入示意图

4.2.2.1 定时器控制块

在 RT-Thread 操作系统中, 定时器控制块由结构体 `struct rt_timer` 定义并形成定时器内核对象, 再链接到内核对象容器中进行管理。它是操作系统用于管理定时器的一个数据结构, 会存储定时器的一些信息, 例如初始节拍数, 超时时的节拍数, 也包含定时器与定时器之间连接用的链表结构, 超时回调函数等。

```
struct rt_timer
{
    struct rt_object parent;
    rt_list_t row[RT_TIMER_SKIP_LIST_LEVEL]; /* 定时器链表节点 */

    void (*timeout_func)(void *parameter); /* 定时器超时调用的函数 */
    void      *parameter;                 /* 超时函数的参数 */
    rt_tick_t init_tick;                /* 定时器初始超时时节拍数 */
    rt_tick_t timeout_tick;              /* 定时器实际超时时的节拍数 */
};

typedef struct rt_timer *rt_timer_t;
```

定时器控制块由 `struct rt_timer` 结构体定义并形成定时器内核对象, 再链接到内核对象容器中进行管理, `list` 成员则用于把一个激活的(已经启动的)定时器链接到 `rt_timer_list` 链表中。

4.2.2.2 定时器跳表 (Skip List) 算法

在前面介绍定时器的工作方式的时候说过, 系统新创建并激活的定时器都会按照以超时时间排序的方式插入到 `rt_timer_list` 链表中, 也就是说 `rt_timer_list` 链表是一个有序链表, RT-Thread 中使用了跳表算法来加快搜索链表元素的速度。

跳表是一种基于并联链表的数据结构, 实现简单, 插入、删除、查找的时间复杂度均为 $O(\log n)$ 。跳表是链表的一种, 但它在链表的基础上增加了“跳跃”功能, 正是这个功能, 使得在查找元素时, 跳表能够提供 $O(\log n)$ 的时间复杂度, 举例如下:

一个有序的链表, 如下图所示, 从该有序链表中搜索元素 {13, 39}, 需要比较的次数分别为 {3, 5}, 总共比较的次数为 $3 + 5 = 8$ 次。



图 4.4: 有序链表示意图

使用跳表算法后可以采用类似二叉搜索树的方法，把一些节点提取出来作为索引，得到如下图所示的结构：

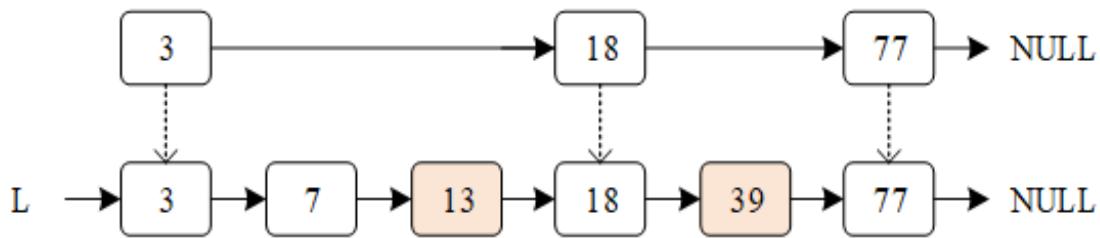


图 4.5: 有序链表索引示意图

在这个结构里把 {3, 18, 77} 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了，例如在搜索 39 时仅比较了 3 次（通过比较 3, 18, 39）。当然我们还可以再从一级索引提取一些元素出来，作为二级索引，这样更能加快元素搜索。

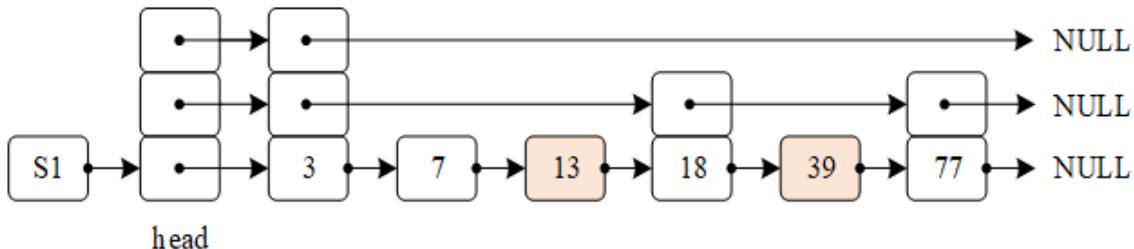


图 4.6: 三层跳表示意图

所以，定时器跳表可以通过上层的索引，在搜索的时候就减少比较次数，提升查找的效率，这是一种通过“空间来换取时间”的算法，在 RT-Thread 中通过宏定义 RT_TIMER_SKIP_LIST_LEVEL 来配置跳表的层数，默认为 1，表示采用一级有序链表图的有序链表算法，每增加一，表示在原链表基础上增加一级索引。

4.2.3 定时器的管理方式

前面介绍了 RT-Thread 定时器并对定时器的工作机制进行了概念上的讲解，本节将深入到定时器的各个接口，帮助读者在代码层次上理解 RT-Thread 定时器。

在系统启动时需要初始化定时器管理系统。可以通过下面的函数接口完成：

```
void rt_system_timer_init(void);
```

如果需要使用 SOFT_TIMER，则系统初始化时，应该调用下面这个函数接口：

```
void rt_system_timer_thread_init(void);
```

定时器控制块中含有定时器相关的重要参数，在定时器各种状态间起到纽带的作用。定时器的相关操作如下图所示，对定时器的操作包含：创建 / 初始化定时器、启动定时器、运行定时器、删除 / 脱离定时器，所有定时器在定时超时后都会从定时器链表中被移除，而周期性定时器会在它再次启动时被加入定时器链表，这与定时器参数设置相关。在每次的操作系统时钟中断发生时，都会对已经超时的定时器状态参数做改变。

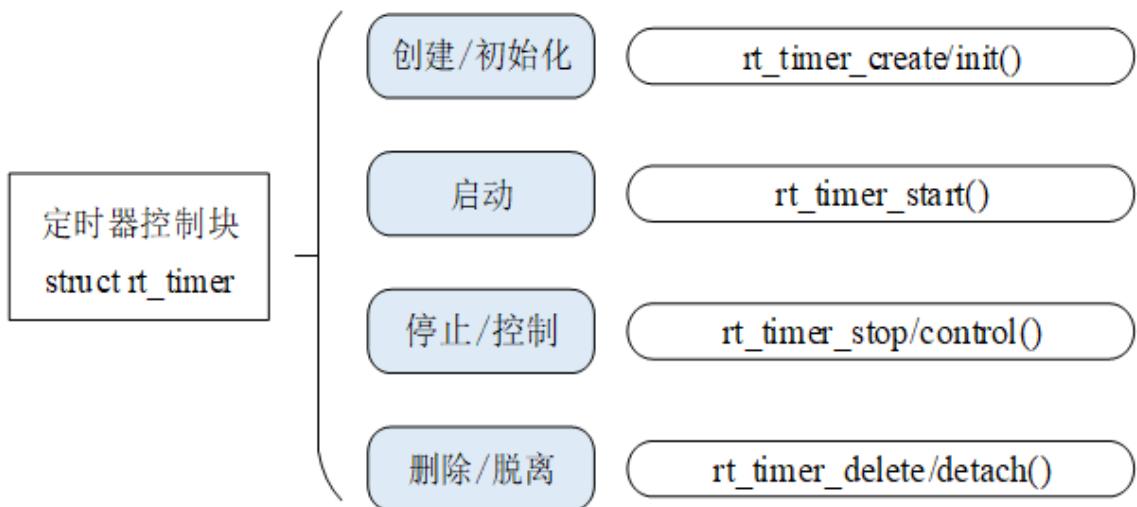


图 4.7: 定时器相关操作

4.2.3.1 创建和删除定时器

当动态创建一个定时器时，可使用下面的函数接口：

```
rt_timer_t rt_timer_create(const char* name,
                           void (*timeout)(void* parameter),
                           void* parameter,
                           rt_tick_t time,
                           rt_uint8_t flag);
```

调用该函数接口后，内核首先从动态内存堆中分配一个定时器控制块，然后对该控制块进行基本的初始化。其中的各参数和返回值说明详见下表：

`rt_timer_create()` 的输入参数和返回值

参数	描述
name	定时器的名称

参数	描述
void (timeout) (void parameter)	定时器超时函数指针（当定时器超时时，系统会调用这个函数）
parameter	定时器超时函数的入口参数（当定时器超时时，调用超时回调函数会把这个参数做为入口参数传递给超时函数）
time	定时器的超时时间，单位是时钟节拍
flag	定时器创建时的参数，支持的值包括单次定时、周期定时、硬件定时器、软件定时器等（可以用“或”关系取多个值）
返回	—
RT_NULL	创建失败（通常会由于系统内存不够用而返回 RT_NULL）
定时器的句柄	定时器创建成功

include/rtdef.h 中定义了一些定时器相关的宏，如下：

```
#define RT_TIMER_FLAG_ONE_SHOT      0x0      /* 单次定时      */
#define RT_TIMER_FLAG_PERIODIC       0x2      /* 周期定时      */

#define RT_TIMER_FLAG_HARD_TIMER    0x0      /* 硬件定时器      */
#define RT_TIMER_FLAG_SOFT_TIMER    0x4      /* 软件定时器      */
```

上面 2 组值可以以“或”逻辑的方式赋给 flag。当指定的 flag 为 RT_TIMER_FLAG_HARD_TIMER 时，如果定时器超时，定时器的回调函数将在时钟中断的服务例程上下文中被调用；当指定的 flag 为 RT_TIMER_FLAG_SOFT_TIMER 时，如果定时器超时，定时器的回调函数将在系统时钟 timer 线程的上下文中被调用。

系统不再使用动态定时器时，可使用下面的函数接口：

```
rt_err_t rt_timer_delete(rt_timer_t timer);
```

调用这个函数接口后，系统会把这个定时器从 rt_timer_list 链表中删除，然后释放相应的定时器控制块占有的内存，其中的各参数和返回值说明详见下表：

rt_timer_delete() 的输入参数和返回值

参数	描述
timer	定时器句柄，指向要删除的定时器
返回	—
RT_EOK	删除成功（如果参数 timer 句柄是一个 RT_NULL，将会导致一个 ASSERT 断言）

4.2.3.2 初始化和脱离定时器

当选择静态创建定时器时，可利用 rt_timer_init 接口来初始化该定时器，函数接口如下：

```
void rt_timer_init(rt_timer_t timer,
                   const char* name,
                   void (*timeout)(void* parameter),
                   void* parameter,
                   rt_tick_t time, rt_uint8_t flag);
```

使用该函数接口时会初始化相应的定时器控制块，初始化相应的定时器名称，定时器超时函数等等，其中的各参数和返回值说明详见下表：

`rt_timer_init()` 的输入参数和返回值

参数	描述
timer	定时器句柄，指向要初始化的定时器控制块
name	定时器的名称
void (timeout) (void parameter)	定时器超时函数指针（当定时器超时时，系统会调用这个函数）
parameter	定时器超时函数的入口参数（当定时器超时时，调用超时回调函数会把这个参数做为入口参数传递给超时函数）
time	定时器的超时时间，单位是时钟节拍
flag	定时器创建时的参数，支持的值包括单次定时、周期定时、硬件定时器、软件定时器（可以用“或”关系取多个值），详见创建定时器小节

当一个静态定时器不需要再使用时，可以使用下面的函数接口：

```
rt_err_t rt_timer_detach(rt_timer_t timer);
```

脱离定时器时，系统会把定时器对象从内核对象容器中脱离，但是定时器对象所占有的内存不会被释放，其中的各参数和返回值说明详见表下表：

`rt_timer_detach()` 的输入参数和返回值

参数	描述
timer	定时器句柄，指向要脱离的定时器控制块
返回	—
RT_EOK	脱离成功

4.2.3.3 启动和停止定时器

当定时器被创建或者初始化以后，并不会被立即启动，必须在调用启动定时器函数接口后，才开始工作，启动定时器函数接口如下：

```
rt_err_t rt_timer_start(rt_timer_t timer);
```

调用定时器启动函数接口后，定时器的状态将更改为激活状态（RT_TIMER_FLAG_ACTIVATED），并按照超时顺序插入到 `rt_timer_list` 队列链表中，其中的各参数和返回值说明详见下表：

`rt_timer_start()` 的输入参数和返回值

参数	描述
<code>timer</code>	定时器句柄，指向要启动的定时器控制块
返回	—
<code>RT_EOK</code>	启动成功

启动定时器的例子请参考后面的示例代码。

启动定时器以后，若想使它停止，可以使用下面的函数接口：

```
rt_err_t rt_timer_stop(rt_timer_t timer);
```

调用定时器停止函数接口后，定时器状态将更改为停止状态，并从 `rt_timer_list` 链表中脱离出来不参与定时器超时检查。当一个（周期性）定时器超时时，也可以调用这个函数接口停止这个（周期性）定时器本身，其中的各参数和返回值说明详见下表：

`rt_timer_stop()` 的输入参数和返回值

参数	描述
<code>timer</code>	定时器句柄，指向要停止的定时器控制块
返回	—
<code>RT_EOK</code>	成功停止定时器
- <code>RT_ERROR</code>	<code>timer</code> 已经处于停止状态

4.2.3.4 控制定时器

除了上述提供的一些编程接口，RT-Thread 也额外提供了定时器控制函数接口，以获取或设置更多定时器的信息。控制定时器函数接口如下：

```
rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg);
```

控制定时器函数接口可根据命令类型参数，来查看或改变定时器的设置，其中的各参数和返回值说明详见下表：

`rt_timer_control()` 的输入参数和返回值

参数	描述
<code>timer</code>	定时器句柄，指向要停止的定时器控制块
<code>cmd</code>	用于控制定时器的命令，当前支持四个命令，分别是设置定时时间，查看定时时间，设置单次触发，设置周期触发

参数	描述
arg	与 cmd 相对应的控制命令参数比如，cmd 为设定超时时间时，就可以将超时时间参数通过 arg 进行设定
返回	—
RT_EOK	成功

函数参数 cmd 支持的命令：

#define RT_TIMER_CTRL_SET_TIME	0x0	/* 设置定时器超时时间 */
#define RT_TIMER_CTRL_GET_TIME	0x1	/* 获得定时器超时时间 */
#define RT_TIMER_CTRL_SET_ONESHOT	0x2	/* 设置定时器为单次定时器 */
#define RT_TIMER_CTRL_SET_PERIODIC	0x3	/* 设置定时器为周期型定时器 */

使用定时器控制接口的代码请见动态定时器例程。

4.3 定时器应用示例

这是一个创建定时器的例子，这个例程会创建两个动态定时器，一个是单次定时，一个是周期性定时并让周期定时器运行一段时间后停止运行，如下所示：

```
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_timer_t timer2;
static int cnt = 0;

/* 定时器 1 超时函数 */
static void timeout1(void *parameter)
{
    rt_kprintf("periodic timer is timeout %d\n", cnt);

    /* 运行第 10 次，停止周期定时器 */
    if (cnt++ >= 9)
    {
        rt_timer_stop(timer1);
        rt_kprintf("periodic timer was stopped! \n");
    }
}

/* 定时器 2 超时函数 */
static void timeout2(void *parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}
```

```

int timer_sample(void)
{
    /* 创建定时器 1 周期定时器 */
    timer1 = rt_timer_create("timer1", timeout1,
                            RT_NULL, 10,
                            RT_TIMER_FLAG_PERIODIC);

    /* 启动定时器 1 */
    if (timer1 != RT_NULL) rt_timer_start(timer1);

    /* 创建定时器 2 单次定时器 */
    timer2 = rt_timer_create("timer2", timeout2,
                            RT_NULL, 30,
                            RT_TIMER_FLAG_ONE_SHOT);

    /* 启动定时器 2 */
    if (timer2 != RT_NULL) rt_timer_start(timer2);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(timer_sample, timer sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >timer_sample
msh >periodic timer is timeout 0
periodic timer is timeout 1
one shot timer is timeout
periodic timer is timeout 2
periodic timer is timeout 3
periodic timer is timeout 4
periodic timer is timeout 5
periodic timer is timeout 6
periodic timer is timeout 7
periodic timer is timeout 8
periodic timer is timeout 9
periodic timer was stopped!

```

周期性定时器 1 的超时函数，每 10 个 OS Tick 运行 1 次，共运行 10 次（10 次后调用 `rt_timer_stop` 使定时器 1 停止运行）；单次定时器 2 的超时函数在第 30 个 OS Tick 时运行一次。

初始化定时器的例子与创建定时器的例子类似，这个程序会初始化 2 个静态定时器，一个是单次定时，一个是周期性的定时，如下代码所示：

初始化静态定时器例程

```
#include <rtthread.h>

/* 定时器的控制块 */
static struct rt_timer timer1;
static struct rt_timer timer2;
static int cnt = 0;

/* 定时器 1 超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");
    /* 运行 10 次 */
    if (cnt++ >= 9)
    {
        rt_timer_stop(&timer1);
    }
}

/* 定时器 2 超时函数 */
static void timeout2(void* parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}

int timer_static_sample(void)
{
    /* 初始化定时器 */
    rt_timer_init(&timer1, "timer1", /* 定时器名字是 timer1 */
                  timeout1, /* 超时时回调的处理函数 */
                  RT_NULL, /* 超时函数的入口参数 */
                  10, /* 定时长度, 以 OS Tick 为单位, 即 10 个 OS Tick */
                  RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
    rt_timer_init(&timer2, "timer2", /* 定时器名字是 timer2 */
                  timeout2, /* 超时时回调的处理函数 */
                  RT_NULL, /* 超时函数的入口参数 */
                  30, /* 定时长度为 30 个 OS Tick */
                  RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

    /* 启动定时器 */
    rt_timer_start(&timer1);
    rt_timer_start(&timer2);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(timer_static_sample, timer_static_sample);
```

仿真运行结果如下：

```
\ | /
- RT -      Thread Operating System
 / | \    3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >timer_static_sample
msh >periodic timer is timeout
periodic timer is timeout
one shot timer is timeout
periodic timer is timeout
```

周期性定时器 1 的超时函数，每 10 个 OS Tick 运行 1 次，共运行 10 次（10 次后调用 `rt_timer_stop` 使定时器 1 停止运行）；单次定时器 2 的超时函数在第 30 个 OS Tick 时运行一次。

4.4 高精度延时

RT-Thread 定时器的最小精度是由系统时钟节拍所决定的（1 OS Tick = 1/RT_TICK_PER_SECOND 秒，`RT_TICK_PER_SECOND` 值在 `rtconfig.h` 文件中定义），定时器设定的时间必须是 OS Tick 的整数倍。当需要实现更短时间长度的系统定时，例如 OS Tick 是 10ms，而程序需要实现 1ms 的定时或延时，这种时候操作系统定时器将不能够满足要求，只能通过读取系统某个硬件定时器的计数器或直接使用硬件定时器的方式。

在 Cortex-M 系列中，SysTick 已经被 RT-Thread 用于作为 OS Tick 使用，它被配置成 1/`RT_TICK_PER_SECOND` 秒后触发一次中断的方式，中断处理函数使用 Cortex-M3 默认的 `SysTick_Handler` 名字。在 Cortex-M3 的 CMSIS (Cortex Microcontroller Software Interface Standard) 规范中规定了 `SystemCoreClock` 代表芯片的主频，所以基于 SysTick 以及 `SystemCoreClock`，我们能够使用 SysTick 获得一个精确的延时函数，如下例所示，Cortex-M3 上的基于 SysTick 的精确延时（需要系统在使能 SysTick 后使用）：

高精度延时的例程如下所示：

```
#include <board.h>
void rt_hw_us_delay(rt_uint32_t us)
{
    rt_uint32_t delta;
    /* 获得延时经过的 tick 数 */
    us = us * (SysTick->LOAD/(1000000/RT_TICK_PER_SECOND));
    /* 获得当前时间 */
    delta = SysTick->VAL;
    /* 循环获得当前时间，直到达到指定的时间后退出循环 */
    while (delta - SysTick->VAL < us);
```

{}

其中入口参数 `us` 指示出需要延时的微秒数目，这个函数只能支持低于 1 OS Tick 的延时，否则 SysTick 会出现溢出而不能够获得指定的延时时间。

第 5 章

线程间同步

在多线程实时系统中，一项工作的完成往往可以通过多个线程协调的方式共同来完成，那么多个线程之间如何“默契”协作才能使这项工作无差错执行？下面举个例子说明。

例如一项工作中的两个线程：一个线程从传感器中接收数据并且将数据写到共享内存中，同时另一个线程周期性的从共享内存中读取数据并发送去显示，下图描述了两个线程间的数据传递：

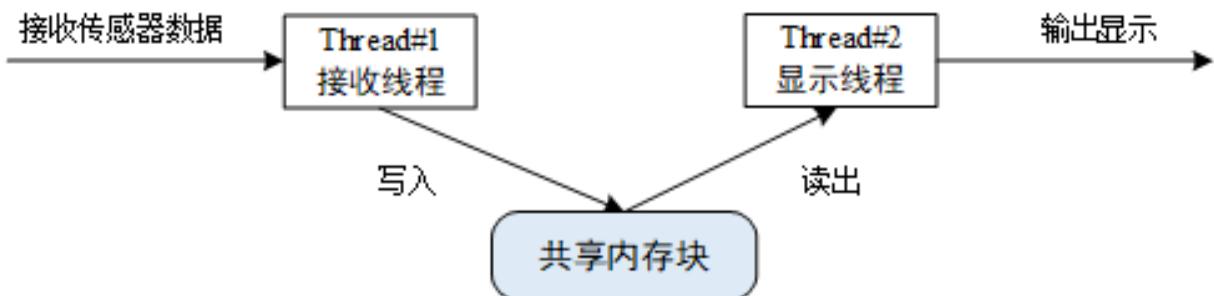


图 5.1: 线程间数据传递示意图

如果对共享内存的访问不是排他性的，那么各个线程间可能同时访问它，这将引起数据一致性的问题。例如，在显示线程试图显示数据之前，接收线程还未完成数据的写入，那么显示将包含不同时间采样的数据，造成显示数据的错乱。

将传感器数据写入到共享内存块的接收线程 #1 和将传感器数据从共享内存块中读出的线程 #2 都会访问同一块内存。为了防止出现数据的差错，两个线程访问的动作必须是互斥进行的，应该是在一个线程对共享内存块操作完成后，才允许另一个线程去操作，这样，接收线程 #1 与显示线程 #2 才能正常配合，使此项工作正确地执行。

同步是指按预定的先后次序进行运行，线程同步是指多个线程通过特定的机制（如互斥量，事件对象，临界区）来控制线程之间的执行顺序，也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间将是无序的。

多个线程操作 / 访问同一块区域（代码），这块代码就称为临界区，上述例子中的共享内存块就是临界区。线程互斥是指对于临界区资源访问的排它性。当多个线程都要使用临界区资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程的同步方式有很多种，其核心思想都是：在访问临界区的时候只允许一个（或一类）线程运行。进入 / 退出临界区的方式有很多种：

1) 调用 `rt_hw_interrupt_disable()` 进入临界区，调用 `rt_hw_interrupt_enable()` 退出临界区；详见《中断管理》的全局中断开关内容。

2) 调用 `rt_enter_critical()` 进入临界区，调用 `rt_exit_critical()` 退出临界区。

本章将介绍多种同步方式：信号量（semaphore）、互斥量（mutex）和事件集（event）。学习完本章，大家将学会如何使用信号量、互斥量、事件集这些对象进行线程间的同步。

5.1 信号量

以生活中的停车场为例来理解信号量的概念：

□ 当停车场空的时候，停车场的管理员发现有很多空车位，此时会让外面的车陆续进入停车场获得停车位；

□ 当停车场的车位满的时候，管理员发现已经没有空车位，将禁止外面的车进入停车场，车辆在外排队等候；

□ 当停车场内有车离开时，管理员发现有空的车位让出，允许外面的车进入停车场；待空车位填满后，又禁止外部车辆进入。

在此例子中，管理员就相当于信号量，管理员手中空车位的个数就是信号量的值（非负数，动态变化）；停车位相当于公共资源（临界区），车辆相当于线程。车辆通过获得管理员的允许取得停车位，就类似于线程通过获得信号量访问公共资源。

5.1.1 信号量工作机制

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。

信号量工作示意图如下图所示，每个信号量对象都有一个信号量值和一个线程等待队列，信号量的值对应了信号量对象的实例数目、资源数目，假如信号量值为 5，则表示共有 5 个信号量实例（资源）可以被使用，当信号量实例数目为零时，再申请该信号量的线程就会被挂起在该信号量的等待队列上，等待可用的信号量实例（资源）。

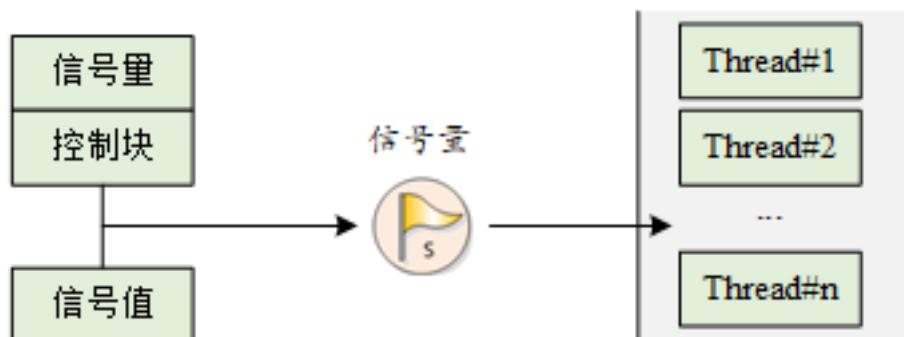


图 5.2: 信号量工作示意图

5.1.2 信号量控制块

在 RT-Thread 中，信号量控制块是操作系统用于管理信号量的一个数据结构，由结构体 `struct rt_semaphore` 表示。另外一种 C 表达方式 `rt_sem_t`，表示的是信号量的句柄，在 C 语言中的实现是指向信号量控制块的指针。信号量控制块结构的详细定义如下：

```
struct rt_semaphore
{
    struct rt_ipc_object parent; /* 继承自 ipc_object 类 */
    rt_uint16_t value;           /* 信号量的值 */
};

/* rt_sem_t 是指向 semaphore 结构体的指针类型 */
typedef struct rt_semaphore* rt_sem_t;
```

`rt_semaphore` 对象从 `rt_ipc_object` 中派生，由 IPC 容器所管理，信号量的最大值是 65535。

5.1.3 信号量的管理方式

信号量控制块中含有信号量相关的重要参数，在信号量各种状态间起到纽带的作用。信号量相关接口如下图所示，对一个信号量的操作包含：创建 / 初始化信号量、获取信号量、释放信号量、删除 / 脱离信号量。

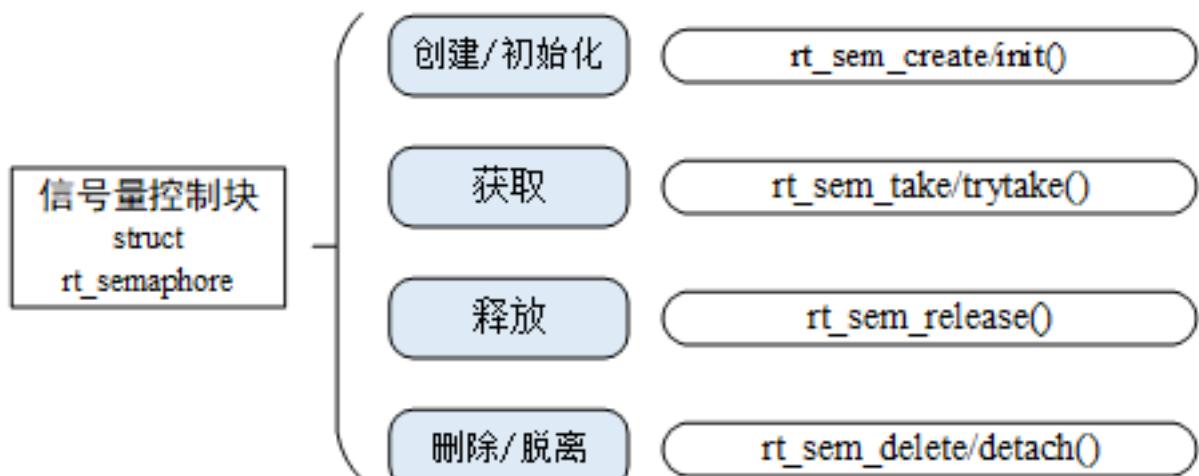


图 5.3: 信号量相关接口

5.1.3.1 创建和删除信号量

当创建一个信号量时，内核首先创建一个信号量控制块，然后对该控制块进行基本的初始化工作，创建信号量使用下面的函数接口：

```
rt_sem_t rt_sem_create(const char *name,
                      rt_uint32_t value,
                      rt_uint8_t flag);
```

当调用这个函数时，系统将先从对象管理器中分配一个 `semaphore` 对象，并初始化这个对象，然后初始化父类 IPC 对象以及与 `semaphore` 相关的部分。在创建信号量指定的参数中，信号量标志参数决定了

当信号量不可用时，多个线程等待的排队方式。当选择 RT_IPC_FLAG_FIFO（先进先出）方式时，那么等待线程队列将按照先进先出的方式排队，先进入的线程将先获得等待的信号量；当选择 RT_IPC_FLAG_PRIO（优先级等待）方式时，等待线程队列将按照优先级进行排队，优先级高的等待线程将先获得等待的信号量。下表描述了该函数的输入参数与返回值：

`rt_sem_create()` 的输入参数和返回值

参数	描述
<code>name</code>	信号量名称
<code>value</code>	信号量初始值
<code>flag</code>	信号量标志，它可以取如下数值： RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
<code>RT_NULL</code>	创建失败
信号量的控制块指针	创建成功

系统不再使用信号量时，可通过删除信号量以释放系统资源，适用于动态创建的信号量。删除信号量使用下面的函数接口：

```
rt_err_t rt_sem_delete(rt_sem_t sem);
```

调用这个函数时，系统将删除这个信号量。如果删除该信号量时，有线程正在等待该信号量，那么删除操作会先唤醒等待在该信号量上的线程（等待线程的返回值是 - RT_ERROR），然后再释放信号量的内存资源。下表描述了该函数的输入参数与返回值：

`rt_sem_delete()` 的输入参数和返回值

参数	描述
<code>sem</code>	<code>rt_sem_create()</code> 创建的信号量对象
返回	——
<code>RT_EOK</code>	删除成功

5.1.3.2 初始化和脱离信号量

对于静态信号量对象，它的内存空间在编译时期就被编译器分配出来，放在读写数据段或未初始化数据段上，此时使用信号量就不再需要使用 `rt_sem_create` 接口来创建它，而只需在使用前对它进行初始化即可。初始化信号量对象可使用下面的函数接口：

```
rt_err_t rt_sem_init(rt_sem_t      sem,
                     const char    *name,
                     rt_uint32_t   value,
                     rt_uint8_t    flag)
```

当调用这个函数时，系统将对这个 `semaphore` 对象进行初始化，然后初始化 IPC 对象以及与 `semaphore` 相关的部分。信号量标志可用上面创建信号量函数里提到的标志。下表描述了该函数的输入参数与返回值：

`rt_sem_init()` 的输入参数和返回值

参数	描述
<code>sem</code>	信号量对象的句柄
<code>name</code>	信号量名称
<code>value</code>	信号量初始值
<code>flag</code>	信号量标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
<code>RT_EOK</code>	初始化成功

脱离信号量就是让信号量对象从内核对象管理器中脱离，适用于静态初始化的信号量。脱离信号量使用下面的函数接口：

```
rt_err_t rt_sem_detach(rt_sem_t sem);
```

使用该函数后，内核先唤醒所有挂在该信号量等待队列上的线程，然后将该信号量从内核对象管理器中脱离。原来挂起在信号量上的等待线程将获得 - RT_ERROR 的返回值。下表描述了该函数的输入参数与返回值：

`rt_sem_detach()` 的输入参数和返回值

参数	描述
<code>sem</code>	信号量对象的句柄
返回	—
<code>RT_EOK</code>	脱离成功

5.1.3.3 获取信号量

线程通过获取信号量来获得信号量资源实例，当信号量值大于零时，线程将获得信号量，并且相应的信号量值会减 1，获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32_t time);
```

在调用这个函数时，如果信号量的值等于零，那么说明当前信号量资源实例不可用，申请该信号量的线程将根据 `time` 参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程或中断释放该信号量。如果在参数 `time` 指定的时间内依然得不到信号量，线程将超时返回，返回值是 - RT_ETIMEOUT。下表描述了该函数的输入参数与返回值：

`rt_sem_take()` 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
time	指定的等待时间，单位是操作系统时钟节拍（OS Tick）
返回	—
RT_EOK	成功获得信号量
-RTETIMEOUT	超时依然未获得信号量
-RT_ERROR	其他错误

5.1.3.4 无等待获取信号量

当用户不想在申请的信号量上挂起线程进行等待时，可以使用无等待方式获取信号量，无等待获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_trytake(rt_sem_t sem);
```

这个函数与 `rt_sem_take(sem, 0)` 的作用相同，即当线程申请的信号量资源实例不可用的时候，它不会等待在该信号量上，而是直接返回 `-RTETIMEOUT`。下表描述了该函数的输入参数与返回值：

`rt_sem_trytake()` 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
返回	—
RT_EOK	成功获得信号量
-RTETIMEOUT	获取失败

5.1.3.5 释放信号量

释放信号量可以唤醒挂起在该信号量上的线程。释放信号量使用下面的函数接口：

```
rt_err_t rt_sem_release(rt_sem_t sem);
```

例如当信号量的值等于零时，并且有线程等待这个信号量时，释放信号量将唤醒等待在该信号量线程队列中的第一个线程，由它获取信号量；否则将把信号量的值加 1。下表描述了该函数的输入参数与返回值：

`rt_sem_release()` 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
返回	—

参数	描述
RT_EOK	成功释放信号量

5.1.4 信号量应用示例

这是一个信号量使用例程，该例程创建了一个动态信号量，初始化两个线程，一个线程发送信号量，一个线程接收到信号量后，执行相应的操作。如下代码所示：

信号量的使用

```
#include <rtthread.h>

#define THREAD_PRIORITY          25
#define THREAD_TIMESLICE         5

/* 指向信号量的指针 */
static rt_sem_t dynamic_sem = RT_NULL;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
static void rt_thread1_entry(void *parameter)
{
    static rt_uint8_t count = 0;

    while(1)
    {
        if(count <= 100)
        {
            count++;
        }
        else
            return;

        /* count 每计数 10 次，就释放一次信号量 */
        if(0 == (count % 10))
        {
            rt_kprintf("t1 release a dynamic semaphore.\n");
            rt_sem_release(dynamic_sem);
        }
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread2_entry(void *parameter)
```

```
{  
    static rt_err_t result;  
    static rt_uint8_t number = 0;  
    while(1)  
    {  
        /* 永久方式等待信号量，获取到信号量，则执行 number 自加的操作 */  
        result = rt_sem_take(dynamic_sem, RT_WAITING_FOREVER);  
        if (result != RT_EOK)  
        {  
            rt_kprintf("t2 take a dynamic semaphore, failed.\n");  
            rt_sem_delete(dynamic_sem);  
            return;  
        }  
        else  
        {  
            number++;  
            rt_kprintf("t2 take a dynamic semaphore. number = %d\n", number);  
        }  
    }  
  
/* 信号量示例的初始化 */  
int semaphore_sample(void)  
{  
    /* 创建一个动态信号量，初始值是 0 */  
    dynamic_sem = rt_sem_create("dsem", 0, RT_IPC_FLAG_FIFO);  
    if (dynamic_sem == RT_NULL)  
    {  
        rt_kprintf("create dynamic semaphore failed.\n");  
        return -1;  
    }  
    else  
    {  
        rt_kprintf("create done. dynamic semaphore value = 0.\n");  
    }  
  
    rt_thread_init(&thread1,  
                  "thread1",  
                  rt_thread1_entry,  
                  RT_NULL,  
                  &thread1_stack[0],  
                  sizeof(thread1_stack),  
                  THREAD_PRIORITY, THREAD_TIMESLICE);  
    rt_thread_startup(&thread1);  
  
    rt_thread_init(&thread2,  
                  "thread2",  
                  rt_thread2_entry,  
                  RT_NULL,
```

```

        &thread2_stack[0],
        sizeof(thread2_stack),
        THREAD_PRIORITY-1, THREAD_TIMESLICE);
rt_thread_startup(&thread2);

return 0;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(semaphore_sample, semaphore sample);

```

仿真运行结果：

```

\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >semaphore_sample
create done. dynamic semaphore value = 0.
msh >t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 1
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 2
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 3
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 4
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 5
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 6
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 7
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 8
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 9
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 10

```

如上面运行结果：线程 1 在 count 计数为 10 的倍数时（count 计数为 100 之后线程退出），发送一个信号量，线程 2 在接收信号量后，对 number 进行加 1 操作。

信号量的另一个应用例程如下所示，本例程将使用 2 个线程、3 个信号量实现生产者与消费者的例子。其中：

3 个信号量分别为：
 ① lock：信号量锁的作用，因为 2 个线程都会对同一个数组 array 进行操作，所以该数组是一个共享资源，锁用来保护这个共享资源。
 ② empty：空位个数，初始化为 5 个空位。
 ③ full：满位个数，初始化为 0 个满位。

2 个线程分别为：
 ① 生产者线程：获取到空位后，产生一个数字，循环放入数组中，然后释放一个满位。
 ② 消费者线程：获取到满位后，读取数组内容并相加，然后释放一个空位。

生产者消费者例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      6
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

/* 定义最大 5 个元素能够被产生 */
#define MAXSEM 5

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

/* 指向生产者、消费者在 array 数组中的读写位置 */
static rt_uint32_t set, get;

/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;

/* 生产者线程入口 */
void producer_thread_entry(void *parameter)
{
    int cnt = 0;

    /* 运行 10 次 */
    while (cnt < 10)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改 array 内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set % MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n", array[set % MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_mdelay(20);
    }
}
```

```
    rt_kprintf("the producer exit!\n");
}

/* 消费者线程入口 */
void consumer_thread_entry(void *parameter)
{
    rt_uint32_t sum = 0;

    while (1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get % MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", (get % MAXSEM), array[get % MAXSEM]);
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到 10 个数目，停止，消费者线程相应停止 */
        if (get == 10) break;

        /* 暂停一小会时间 */
        rt_thread_mdelay(50);
    }

    rt_kprintf("the consumer sum is: %d\n", sum);
    rt_kprintf("the consumer exit!\n");
}

int producer_consumer(void)
{
    set = 0;
    get = 0;

    /* 初始化 3 个信号量 */
    rt_sem_init(&sem_lock, "lock", 1, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty", MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full, "full", 0, RT_IPC_FLAG_FIFO);

    /* 创建生产者线程 */
    producer_tid = rt_thread_create("producer",
                                    producer_thread_entry, RT_NULL,
```

```

        THREAD_STACK_SIZE,
        THREAD_PRIORITY - 1,
        THREAD_TIMESLICE);

if (producer_tid != RT_NULL)
{
    rt_thread_startup(producer_tid);
}
else
{
    rt_kprintf("create thread producer failed");
    return -1;
}

/* 创建消费者线程 */
consumer_tid = rt_thread_create("consumer",
                                consumer_thread_entry, RT_NULL,
                                THREAD_STACK_SIZE,
                                THREAD_PRIORITY + 1,
                                THREAD_TIMESLICE);

if (consumer_tid != RT_NULL)
{
    rt_thread_startup(consumer_tid);
}
else
{
    rt_kprintf("create thread consumer failed");
    return -1;
}

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(producer_consumer, producer_consumer_sample);

```

该例程的仿真结果如下：

```

\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >producer_consumer
the producer generates a number: 1
the consumer[0] get a number: 1
msh >the producer generates a number: 2
the producer generates a number: 3
the consumer[1] get a number: 2
the producer generates a number: 4
the producer generates a number: 5

```

```

the producer generates a number: 6
the consumer[2] get a number: 3
the producer generates a number: 7
the producer generates a number: 8
the consumer[3] get a number: 4
the producer generates a number: 9
the consumer[4] get a number: 5
the producer generates a number: 10
the producer exit!
the consumer[0] get a number: 6
the consumer[1] get a number: 7
the consumer[2] get a number: 8
the consumer[3] get a number: 9
the consumer[4] get a number: 10
the consumer sum is: 55
the consumer exit!

```

本例程可以理解为生产者生产产品放入仓库，消费者从仓库中取走产品。

(1) 生产者线程:

- 1) 获取 1 个空位（放产品 number），此时空位减 1；
- 2) 上锁保护；本次的产生的 number 值为 cnt+1，把值循环存入数组 array 中；再开锁；
- 3) 释放 1 个满位（给仓库中放置一个产品，仓库就多一个满位），满位加 1；

(2) 消费者线程:

- 1) 获取 1 个满位（取产品 number），此时满位减 1；
- 2) 上锁保护；将本次生产者生产的 number 值从 array 中读出来，并与上次的 number 值相加；再开锁；
- 3) 释放 1 个空位（从仓库上取走一个产品，仓库就多一个空位），空位加 1。

生产者依次产生 10 个 number，消费者依次取走，并将 10 个 number 的值求和。信号量锁 lock 保护 array 临界区资源：保证了消费者每次取 number 值的排他性，实现了线程间同步。

5.1.5 信号量的使用场合

信号量是一种非常灵活的同步方式，可以运用在多种场合中。形成锁、同步、资源计数等关系，也能方便的用于线程与线程、中断与线程间的同步中。

5.1.5.1 线程同步

线程同步是信号量最简单的一类应用。例如，使用信号量进行两个线程之间的同步，信号量的值初始化成 0，表示具备 0 个信号量资源实例；而尝试获得该信号量的线程，将直接在这个信号量上进行等待。

当持有信号量的线程完成它处理的工作时，释放这个信号量，可以把等待在这个信号量上的线程唤醒，让它执行下一部分工作。这类场合也可以看成把信号量用于工作完成标志：持有信号量的线程完成它自己的工作，然后通知等待该信号量的线程继续下一部分工作。

5.1.5.2 锁

锁，单一的锁常应用于多个线程间对同一共享资源（即临界区）的访问。信号量在作为锁来使用时，通常应将信号量资源实例初始化成 1，代表系统默认有一个资源可用，因为信号量的值始终在 1 和 0 之间变动，所以这类锁也叫做二值信号量。如下图所示，当线程需要访问共享资源时，它需要先获得这个资源锁。当这个线程成功获得资源锁时，其他打算访问共享资源的线程会由于获取不到资源而挂起，这是因为其他线程在试图获取这个锁时，这个锁已经被锁上（信号量值是 0）。当获得信号量的线程处理完毕，退出临界区时，它将会释放信号量并把锁解开，而挂起在锁上的第一个等待线程将被唤醒从而获得临界区的访问权。

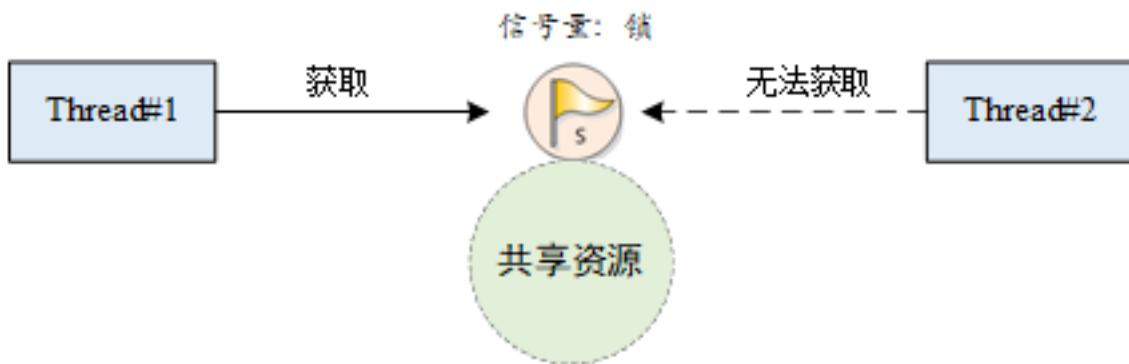


图 5.4: 锁

5.1.5.3 中断与线程的同步

信号量也能够方便地应用于中断与线程间的同步，例如一个中断触发，中断服务例程需要通知线程进行相应的数据处理。这个时候可以设置信号量的初始值是 0，线程在试图持有这个信号量时，由于信号量的初始值是 0，线程直接在这个信号量上挂起直到信号量被释放。当中断触发时，先进行与硬件相关的动作，例如从硬件的 I/O 口中读取相应数据，并确认中断以清除中断源，而后释放一个信号量来唤醒相应的线程以做后续的数据处理。例如 FinSH 线程的处理方式，如下图所示。

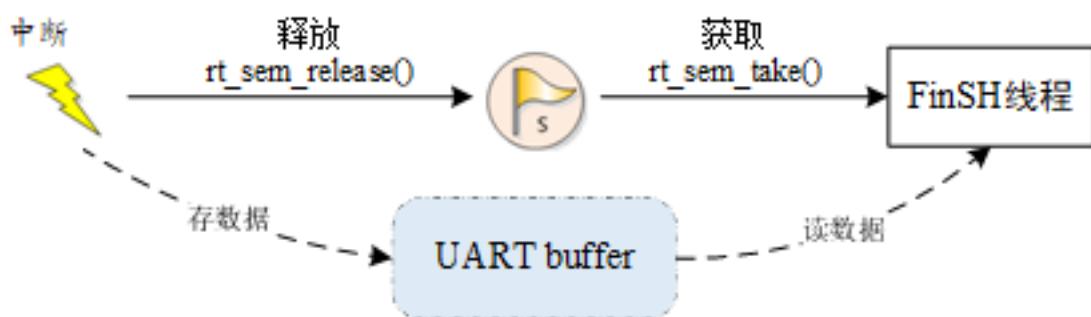


图 5.5: FinSH 的中断、线程间同步示意图

信号量的值初始为 0，当 FinSH 线程试图取得信号量时，因为信号量值是 0，所以它会被挂起。当 console 设备有数据输入时，产生中断，从而进入中断服务例程。在中断服务例程中，它会读取 console 设备的数据，并把读得的数据放入 UART buffer 中进行缓冲，而后释放信号量，释放信号量的操作将唤醒 shell 线程。在中断服务例程运行完毕后，如果系统中没有比 shell 线程优先级更高的就绪线程存在时，shell 线程将持有信号量并运行，从 UART buffer 缓冲区中获取输入的数据。

!!! note “注意事项” 中断与线程间的互斥不能采用信号量（锁）的方式，而应采用开关中断的方式。

5.1.5.4 资源计数

信号量也可以认为是一个递增或递减的计数器，需要注意的是信号量的值非负。例如：初始化一个信号量的值为 5，则这个信号量可最大连续减少 5 次，直到计数器减为 0。资源计数适合于线程间工作处理速度不匹配的场合，这个时候信号量可以做为前一线程工作完成个数的计数，而当调度到后一线程时，它也可以以一种连续的方式一次处理多个事件。例如，生产者与消费者问题中，生产者可以对信号量进行多次释放，而后消费者被调度到时能够一次处理多个信号量资源。

!!! note “注意事项”一般资源计数类型多是混合方式的线程间同步，因为对于单个的资源处理依然存在线程的多重访问，这就需要对一个单独的资源进行访问、处理，并进行锁方式的互斥操作。

5.2 互斥量

互斥量又叫相互排斥的信号量，是一种特殊的二值信号量。互斥量类似于只有一个车位的停车场：当有一辆车进入的时候，将停车场大门锁住，其他车辆在外面等候。当里面的车出来时，将停车场大门打开，下一辆车才可以进入。

5.2.1 互斥量工作机制

互斥量和信号量不同的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的状态只有两种，开锁或闭锁（两种状态值）。当有线程持有它时，互斥量处于闭锁状态，由这个线程获得它的所有权。相反，当这个线程释放它时，将对互斥量进行开锁，失去它的所有权。当一个线程持有互斥量时，其他线程将不能够对它进行开锁或持有它，持有该互斥量的线程也能够再次获得这个锁而不被挂起，如下图所示。这个特性与一般的二值信号量有很大的不同：在信号量中，因为已经不存在实例，线程递归持有会发生主动挂起（最终形成死锁）。

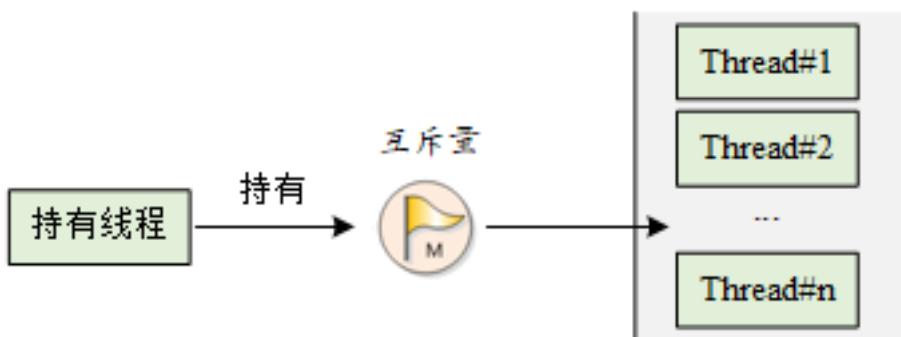


图 5.6: 互斥量工作示意图

使用信号量会导致的另一个潜在问题是线程优先级翻转问题。所谓优先级翻转，即当一个高优先级线程试图通过信号量机制访问共享资源时，如果该信号量已被一低优先级线程持有，而这个低优先级线程在运行过程中可能又被其它一些中等优先级的线程抢占，因此造成高优先级线程被许多具有较低优先级的线程阻塞，实时性难以得到保证。如下图所示：有优先级为 A、B 和 C 的三个线程，优先级 A>B>C。线程 A, B 处于挂起状态，等待某一事件触发，线程 C 正在运行，此时线程 C 开始使用某一共享资源 M。在使用过程中，线程 A 等待的事件到来，线程 A 转为就绪态，因为它比线程 C 优先级高，所以立即执行。但是当线程 A 要使用共享资源 M 时，由于其正在被线程 C 使用，因此线程 A 被挂起切换到线程 C 运行。如果

此时线程 B 等待的事件到来，则线程 B 转为就绪态。由于线程 B 的优先级比线程 C 高，因此线程 B 开始运行，直到其运行完毕，线程 C 才开始运行。只有当线程 C 释放共享资源 M 后，线程 A 才得以执行。在这种情况下，优先级发生了翻转：线程 B 先于线程 A 运行。这样便不能保证高优先级线程的响应时间。

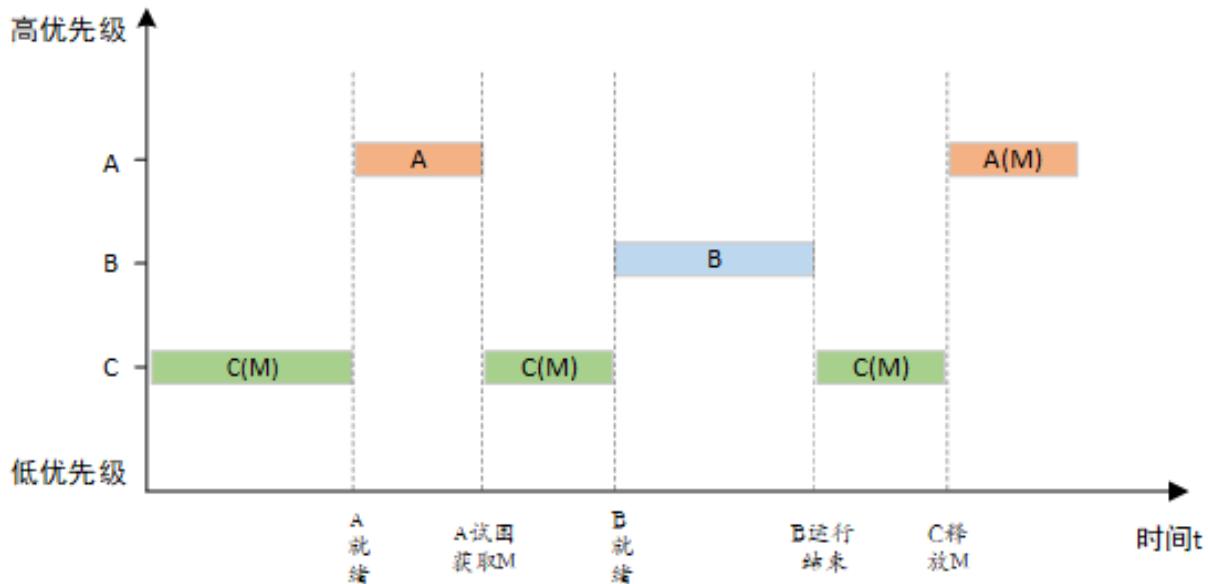


图 5.7: 优先级反转 (M 为信号量)

在 RT-Thread 操作系统中，互斥量可以解决优先级翻转问题，实现的是优先级继承算法。优先级继承是通过在线程 A 尝试获取共享资源而被挂起的期间内，将线程 C 的优先级提升到线程 A 的优先级别，从而解决优先级翻转引起的问题。这样能够防止 C（间接地防止 A）被 B 抢占，如下图所示。优先级继承是指，提高某个占有某种资源的低优先级线程的优先级，使之与所有等待该资源的线程中优先级最高的那个线程的优先级相等，然后执行，而当这个低优先级线程释放该资源时，优先级重新回到初始设定。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

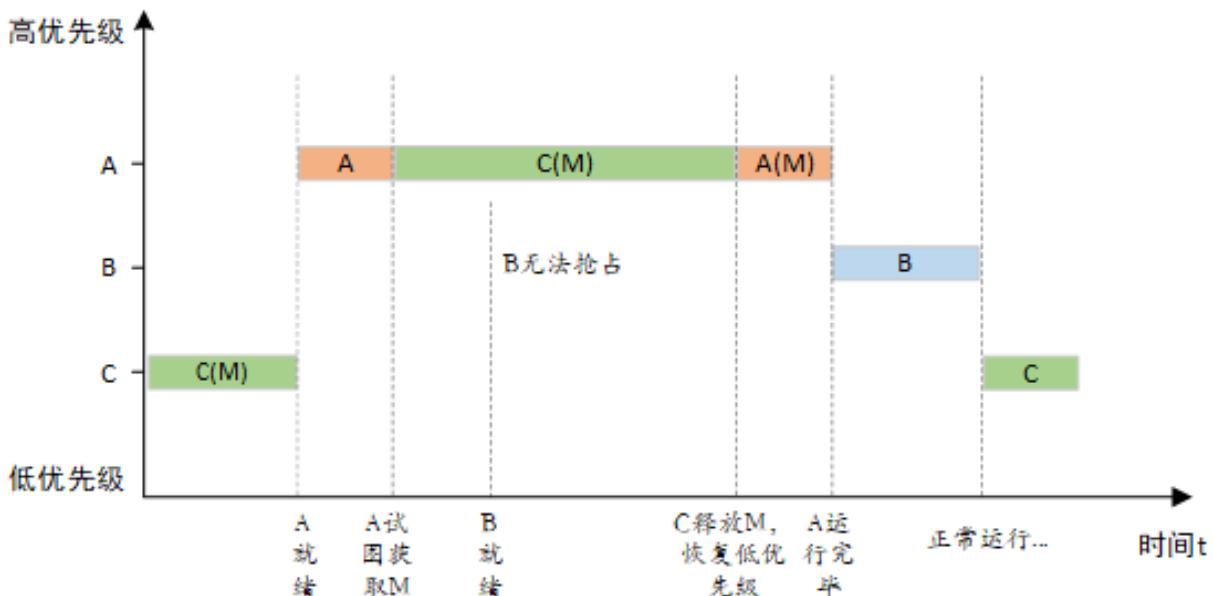


图 5.8: 优先级继承 (M 为互斥量)

!!! note “注意事项” 在获得互斥量后，请尽快释放互斥量，并且在持有互斥量的过程中，不得再行更

改持有互斥量线程的优先级。

5.2.2 互斥量控制块

在 RT-Thread 中，互斥量控制块是操作系统用于管理互斥量的一个数据结构，由结构体 struct rt_mutex 表示。另外一种 C 表达方式 rt_mutex_t，表示的是互斥量的句柄，在 C 语言中的实现是指互斥量控制块的指针。互斥量控制块结构的详细定义请见以下代码：

```
struct rt_mutex
{
    struct rt_ipc_object parent; /* 继承自 ipc_object 类 */

    rt_uint16_t      value;      /* 互斥量的值 */
    rt_uint8_t       original_priority; /* 持有线程的原始优先级 */
    rt_uint8_t       hold;       /* 持有线程的持有次数 */
    struct rt_thread *owner;     /* 当前拥有互斥量的线程 */

};

/* rt_mutex_t 为指向互斥量结构体的指针类型 */
typedef struct rt_mutex* rt_mutex_t;
```

rt_mutex 对象从 rt_ipc_object 中派生，由 IPC 容器所管理。

5.2.3 互斥量的管理方式

互斥量控制块中含有互斥相关的重要参数，在互斥量功能的实现中起到重要的作用。互斥量相关接口如下图所示，对一个互斥量的操作包含：创建 / 初始化互斥量、获取互斥量、释放互斥量、删除 / 脱离互斥量。

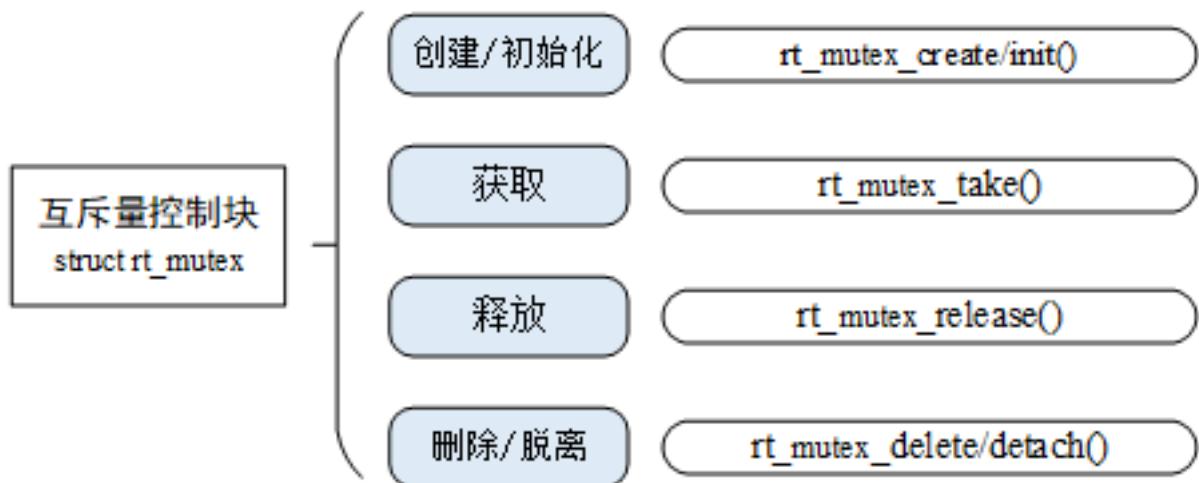


图 5.9: 互斥量相关接口

5.2.3.1 创建和删除互斥量

创建一个互斥量时，内核首先创建一个互斥量控制块，然后完成对该控制块的初始化工作。创建互斥量使用下面的函数接口：

```
rt_mutex_t rt_mutex_create (const char* name, rt_uint8_t flag);
```

可以调用 `rt_mutex_create` 函数创建一个互斥量，它的名字由 `name` 所指定。当调用这个函数时，系统将先从对象管理器中分配一个 `mutex` 对象，并初始化这个对象，然后初始化父类 IPC 对象以及与 mutex 相关的部分。互斥量的 `flag` 标志设置为 `RT_IPC_FLAG_PRIO`，表示在多个线程等待资源时，将由优先级高的线程优先获得资源。`flag` 设置为 `RT_IPC_FLAG_FIFO`，表示在多个线程等待资源时，将按照先来先得的顺序获得资源。下表描述了该函数的输入参数与返回值：

`rt_mutex_create()` 的输入参数和返回值

参数	描述
<code>name</code>	互斥量的名称
<code>flag</code>	互斥量标志，它可以取如下数值： <code>RT_IPC_FLAG_FIFO</code> 或 <code>RT_IPC_FLAG_PRIO</code>
返回	——
互斥量句柄	创建成功
<code>RT_NULL</code>	创建失败

当不再使用互斥量时，通过删除互斥量以释放系统资源，适用于动态创建的互斥量。删除互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_delete (rt_mutex_t mutex);
```

当删除一个互斥量时，所有等待此互斥量的线程都将被唤醒，等待线程获得的返回值是 `- RT_ERROR`。然后系统将该互斥量从内核对象管理器链表中删除并释放互斥量占用的内存空间。下表描述了该函数的输入参数与返回值：

`rt_mutex_delete()` 的输入参数和返回值

参数	描述
<code>mutex</code>	互斥量对象的句柄
返回	——
<code>RT_EOK</code>	删除成功

5.2.3.2 初始化和脱离互斥量

静态互斥量对象的内存是在系统编译时由编译器分配的，一般放于读写数据段或未初始化数据段中。在使用这类静态互斥量对象前，需要先进行初始化。初始化互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8_t flag);
```

使用该函数接口时，需指定互斥量对象的句柄（即指向互斥量控制块的指针），互斥量名称以及互斥量标志。互斥量标志可用上面创建互斥量函数里提到的标志。下表描述了该函数的输入参数与返回值：

`rt_mutex_init()` 的输入参数和返回值

参数	描述
<code>mutex</code>	互斥量对象的句柄，它由用户提供，并指向互斥量对象的内存块
<code>name</code>	互斥量的名称
<code>flag</code>	互斥量标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
<code>RT_EOK</code>	初始化成功

脱离互斥量将把互斥量对象从内核对象管理器中脱离，适用于静态初始化的互斥量。脱离互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_detach (rt_mutex_t mutex);
```

使用该函数接口后，内核先唤醒所有挂在该互斥量上的线程（线程的返回值是 `-RT_ERROR`），然后系统将该互斥量从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

`rt_mutex_detach()` 的输入参数和返回值

参数	描述
<code>mutex</code>	互斥量对象的句柄
返回	—
<code>RT_EOK</code>	成功

5.2.3.3 获取互斥量

线程获取了互斥量，那么线程就有了对该互斥量的所有权，即某一个时刻一个互斥量只能被一个线程持有。获取互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32_t time);
```

如果互斥量没有被其他线程控制，那么申请该互斥量的线程将成功获得该互斥量。如果互斥量已经被当前线程线程控制，则该互斥量的持有计数加 1，当前线程也不会挂起等待。如果互斥量已经被其他线程占有，则当前线程在该互斥量上挂起等待，直到其他线程释放它或者等待时间超过指定的超时时间。下表描述了该函数的输入参数与返回值：

`rt_mutex_take()` 的输入参数和返回值

参数	描述
<code>mutex</code>	互斥量对象的句柄
<code>time</code>	指定等待的时间
返回	—

参数	描述
RT_EOK	成功获得互斥量
-RTETIMEOUT	超时
-RT_ERROR	获取失败

5.2.3.4 释放互斥量

当线程完成互斥资源的访问后，应尽快释放它占据的互斥量，使得其他线程能及时获取该互斥量。释放互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_release(rt_mutex_t mutex);
```

使用该函数接口时，只有已经拥有互斥量控制权的线程才能释放它，每释放一次该互斥量，它的持有计数就减 1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），它变为可用，等待在该信号量上的线程将被唤醒。如果线程的运行优先级被互斥量提升，那么当互斥量被释放后，线程恢复为持有互斥量前的优先级。下表描述了该函数的输入参数与返回值：

`rt_mutex_release()` 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄
返回	—
RT_EOK	成功

5.2.4 互斥量应用示例

这是一个互斥量的应用例程，互斥锁是一种保护共享资源的方法。当一个线程拥有互斥锁的时候，可以保护共享资源不被其他线程破坏。下面用一个例子来说明，有两个线程：线程 1 和线程 2，线程 1 对 2 个 `number` 分别进行加 1 操作；线程 2 也对 2 个 `number` 分别进行加 1 操作，使用互斥量保证线程改变 2 个 `number` 值的操作不被打断。如下代码所示：

互斥量例程

```
#include <rtthread.h>

#define THREAD_PRIORITY          8
#define THREAD_TIMESLICE         5

/* 指向互斥量的指针 */
static rt_mutex_t dynamic_mutex = RT_NULL;
static rt_uint8_t number1,number2 = 0;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
```

```
static struct rt_thread thread1;
static void rt_thread_entry1(void *parameter)
{
    while(1)
    {
        /* 线程 1 获取到互斥量后，先后对 number1、number2 进行加 1 操作，然后释放
         * 互斥量 */
        rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
        number1++;
        rt_thread_mdelay(10);
        number2++;
        rt_mutex_release(dynamic_mutex);
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread_entry2(void *parameter)
{
    while(1)
    {
        /* 线程 2 获取到互斥量后，检查 number1、number2 的值是否相同，相同则表示
         * mutex 起到了锁的作用 */
        rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
        if(number1 != number2)
        {
            rt_kprintf("not protect.number1 = %d, number2 = %d \n", number1, number2);
            ;
        }
        else
        {
            rt_kprintf("mutex protect ,number1 = number2 is %d\n", number1);
        }

        number1++;
        number2++;
        rt_mutex_release(dynamic_mutex);

        if(number1>=50)
            return;
    }
}

/* 互斥量示例的初始化 */
int mutex_sample(void)
{
    /* 创建一个动态互斥量 */
    dynamic_mutex = rt_mutex_create("dmutex", RT_IPC_FLAG_FIFO);
```

```

if (dynamic_mutex == RT_NULL)
{
    rt_kprintf("create dynamic mutex failed.\n");
    return -1;
}

rt_thread_init(&thread1,
               "thread1",
               rt_thread_entry1,
               RT_NULL,
               &thread1_stack[0],
               sizeof(thread1_stack),
               THREAD_PRIORITY, THREAD_TIMESLICE);
rt_thread_startup(&thread1);

rt_thread_init(&thread2,
               "thread2",
               rt_thread_entry2,
               RT_NULL,
               &thread2_stack[0],
               sizeof(thread2_stack),
               THREAD_PRIORITY-1, THREAD_TIMESLICE);
rt_thread_startup(&thread2);
return 0;
}

/* 导出到 MSH 命令列表中 */
MSH_CMD_EXPORT(mutex_sample, mutex sample);

```

线程 1 与线程 2 中均使用互斥量保护对 2 个 number 的操作（倘若将线程 1 中的获取、释放互斥量语句注释掉，线程 1 将对 number 不再做保护），仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >mutex_sample
msh >mutex protect ,number1 = number2 is 1
mutex protect ,number1 = number2 is 2
mutex protect ,number1 = number2 is 3
mutex protect ,number1 = number2 is 4
...
mutex protect ,number1 = number2 is 48
mutex protect ,number1 = number2 is 49

```

线程使用互斥量保护对两个 number 的操作，使 number 值保持一致。

互斥量的另一个例子见下面的代码，这个例子将创建 3 个动态线程以检查持有互斥量时，持有的线程优先级是否被调整到等待线程优先级中的最高优先级。

防止优先级翻转特性例程

```
#include <rtthread.h>

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

#define THREAD_PRIORITY      10
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    /* 先让低优先级线程运行 */
    rt_thread_mdelay(100);

    /* 此时 thread3 持有 mutex，并且 thread2 等待持有 mutex */

    /* 检查 thread2 与 thread3 的优先级情况 */
    if (tid2->current_priority != tid3->current_priority)
    {
        /* 优先级不相同，测试失败 */
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test failed.\n");
        return;
    }
    else
    {
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test OK.\n");
    }
}

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_err_t result;

    rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);

    /* 先让低优先级线程运行 */
    rt_thread_mdelay(50);
```

```
/*
 * 试图持有互斥锁，此时 thread3 持有，应把 thread3 的优先级提升
 * 到 thread2 相同的优先级
 */
result = rt_mutex_take(mutex, RT_WAITING_FOREVER);

if (result == RT_EOK)
{
    /* 释放互斥锁 */
    rt_mutex_release(mutex);
}

/*
 * 线程 3 入口 */
static void thread3_entry(void *parameter)
{
    rt_tick_t tick;
    rt_err_t result;

    rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);

    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        rt_kprintf("thread3 take a mutex, failed.\n");
    }

    /* 做一个长时间的循环，500ms */
    tick = rt_tick_get();
    while (rt_tick_get() - tick < (RT_TICK_PER_SECOND / 2)) ;

    rt_mutex_release(mutex);
}

int pri_inversion(void)
{
    /* 创建互斥锁 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
    if (mutex == RT_NULL)
    {
        rt_kprintf("create dynamic mutex failed.\n");
        return -1;
    }

    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                           thread1_entry,
                           RT_NULL,
```

```

        THREAD_STACK_SIZE,
        THREAD_PRIORITY - 1, THREAD_TIMESLICE);

if (tid1 != RT_NULL)
    rt_thread_startup(tid1);

/* 创建线程 2 */
tid2 = rt_thread_create("thread2",
                        thread2_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);

/* 创建线程 3 */
tid3 = rt_thread_create("thread3",
                        thread3_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY + 1, THREAD_TIMESLICE);
if (tid3 != RT_NULL)
    rt_thread_startup(tid3);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pri_inversion, prio_inversion sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >pri_inversion
the priority of thread2 is: 10
the priority of thread3 is: 11
the priority of thread2 is: 10
the priority of thread3 is: 10
test OK.

```

例程演示了互斥量的使用方法。线程 3 先持有互斥量，而后线程 2 试图持有互斥量，此时线程 3 的优先级被提升为和线程 2 的优先级相同。

!!! note “注意事项” 需要切记的是互斥量不能在中断服务例程中使用。

5.2.5 互斥量的使用场合

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。互斥量更适合于：

- (1) 线程多次持有互斥量的情况下。这样可以避免同一线程多次递归持有而造成死锁的问题。
- (2) 可能会由于多线程同步而造成优先级翻转的情况。

5.3 事件集

事件集也是线程间同步的机制之一，一个事件集可以包含多个事件，利用事件集可以完成一对多，多对多的线程间同步。下面以坐公交为例说明事件，在公交站等公交时可能有以下几种情况：

- P1 坐公交去某地，只有一种公交可以到达目的地，等到此公交即可出发。
- P1 坐公交去某地，有 3 种公交都可以到达目的地，等到其中任意一辆即可出发。
- P1 约另一人 P2 一起去某地，则 P1 必须要等到“同伴 P2 到达公交站”与“公交到达公交站”两个条件都满足后，才能出发。

这里，可以将 P1 去某地视为线程，将“公交到达公交站”、“同伴 P2 到达公交站”视为事件的发生，情况 □ 是特定事件唤醒线程；情况 □ 是任意单个事件唤醒线程；情况 □ 是多个事件同时发生才唤醒线程。

5.3.1 事件集工作机制

事件集主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程与多个事件的关系可设置为：其中任意一个事件唤醒线程，或几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件。这种多个事件的集合可以用一个 32 位无符号整型变量来表示，变量的每一位代表一个事件，线程通过“逻辑与”或“逻辑或”将一个或多个事件关联起来，形成事件组合。事件的“逻辑或”也称为是独立型同步，指的是线程与任何事件之一发生同步；事件“逻辑与”也称为是关联型同步，指的是线程与若干事件都发生同步。

RT-Thread 定义的事件集有以下特点：

- 1) 事件只与线程相关，事件间相互独立：每个线程可拥有 32 个事件标志，采用一个 32 bit 无符号整型数进行记录，每一个 bit 代表一个事件；
- 2) 事件仅用于同步，不提供数据传输功能；
- 3) 事件无排队性，即多次向线程发送同一事件（如果线程还未读走），其效果等同于只发送一次。

在 RT-Thread 中，每个线程都拥有一个事件信息标记，它有三个属性，分别是 RT_EVENT_FLAG_AND（逻辑与），RT_EVENT_FLAG_OR（逻辑或）以及 RT_EVENT_FLAG_CLEAR（清除标记）。当线程等待事件同步时，可以通过 32 个事件标志和这个事件信息标记来判断当前接收的事件是否满足同步条件。



图 5.10: 事件集工作示意图

如上图所示，线程 #1 的事件标志中第 1 位和第 30 位被置位，如果事件信息标记位设为逻辑与，则表示线程 #1 只有在事件 1 和事件 30 都发生以后才会被触发唤醒，如果事件信息标记位设为逻辑或，则事件 1 或事件 30 中的任意一个发生都会触发唤醒线程 #1。如果信息标记同时设置了清除标记位，则当线程 #1 唤醒后将主动把事件 1 和事件 30 清为零，否则事件标志将依然存在（即置 1）。

5.3.2 事件集控制块

在 RT-Thread 中，事件集控制块是操作系统用于管理事件的一个数据结构，由结构体 struct rt_event 表示。另外一种 C 表达方式 rt_event_t，表示的是事件集的句柄，在 C 语言中的实现是事件集控制块的指针。事件集控制块结构的详细定义请见以下代码：

```
struct rt_event
{
    struct rt_ipc_object parent; /* 继承自 ipc_object 类 */

    /* 事件集合，每一 bit 表示 1 个事件，bit 位的值可以标记某事件是否发生 */
    rt_uint32_t set;
};

/* rt_event_t 是指向事件结构体的指针类型 */
typedef struct rt_event* rt_event_t;
```

rt_event 对象从 rt_ipc_object 中派生，由 IPC 容器所管理。

5.3.3 事件集的管理方式

事件集控制块中含有与事件集相关的重要参数，在事件集功能的实现中起重要的作用。事件集相关接口如下图所示，对一个事件集的操作包含：创建 / 初始化事件集、发送事件、接收事件、删除 / 脱离事件集。

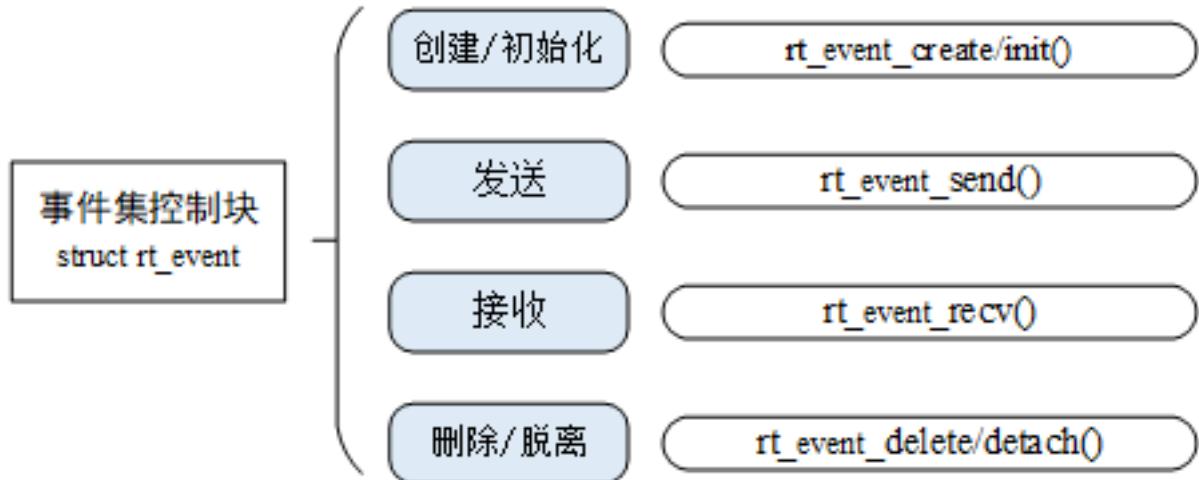


图 5.11: 事件相关接口

5.3.3.1 创建和删除事件集

当创建一个事件集时，内核首先创建一个事件集控制块，然后对该事件集控制块进行基本的初始化，创建事件集使用下面的函数接口：

```
rt_event_t rt_event_create(const char* name, rt_uint8_t flag);
```

调用该函数接口时，系统会从对象管理器中分配事件集对象，并初始化这个对象，然后初始化父类 IPC 对象。下表描述了该函数的输入参数与返回值：

`rt_event_create()` 的输入参数和返回值

参数	描述
name	事件集的名称
flag	事件集的标志，它可以取如下数值： RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
RT_NULL	创建失败
事件对象的句柄	创建成功

系统不再使用 `rt_event_create()` 创建的事件集对象时，通过删除事件集对象控制块来释放系统资源。删除事件集可以使用下面的函数接口：

```
rt_err_t rt_event_delete(rt_event_t event);
```

在调用 `rt_event_delete` 函数删除一个事件集对象时，应该确保该事件集不再被使用。在删除前会唤醒所有挂起在该事件集上的线程（线程的返回值是 - RT_ERROR），然后释放事件集对象占用的内存块。下表描述了该函数的输入参数与返回值：

`rt_event_delete()` 的输入参数和返回值

参数	描述
event	事件集对象的句柄
返回	——
RT_EOK	成功

5.3.3.2 初始化和脱离事件集

静态事件集对象的内存是在系统编译时由编译器分配的，一般放于读写数据段或未初始化数据段中。在使用静态事件集对象前，需要先行对它进行初始化操作。初始化事件集使用下面的函数接口：

```
rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8_t flag);
```

调用该接口时，需指定静态事件集对象的句柄（即指向事件集控制块的指针），然后系统会初始化事件集对象，并加入到系统对象容器中进行管理。下表描述了该函数的输入参数与返回值：

rt_event_init() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
name	事件集的名称
flag	事件集的标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
RT_EOK	成功

系统不再使用 rt_event_init() 初始化的事件集对象时，通过脱离事件集对象控制块来释放系统资源。脱离事件集是将事件集对象从内核对象管理器中脱离。脱离事件集使用下面的函数接口：

```
rt_err_t rt_event_detach(rt_event_t event);
```

用户调用这个函数时，系统首先唤醒所有挂在该事件集等待队列上的线程（线程的返回值是 -RT_ERROR），然后将该事件集从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

rt_event_detach() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
返回	——
RT_EOK	成功

5.3.3.3 发送事件

发送事件函数可以发送事件集中的一个或多个事件，如下：

```
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set);
```

使用该函数接口时，通过参数 `set` 指定的事件标志来设定 `event` 事件集对象的事件标志值，然后遍历等待在 `event` 事件集对象上的等待线程链表，判断是否有线程的事件激活要求与当前 `event` 对象事件标志值匹配，如果有，则唤醒该线程。下表描述了该函数的输入参数与返回值：

`rt_event_send()` 的输入参数和返回值

参数	描述
<code>event</code>	事件集对象的句柄
<code>set</code>	发送的一个或多个事件的标志值
返回	—
<code>RT_EOK</code>	成功

5.3.3.4 接收事件

内核使用 32 位的无符号整数来标识事件集，它的每一位代表一个事件，因此一个事件集对象可同时等待接收 32 个事件，内核可以通过指定选择参数“逻辑与”或“逻辑或”来选择如何激活线程，使用“逻辑与”参数表示只有当所有等待的事件都发生时才激活线程，而使用“逻辑或”参数则表示只要有一个等待的事件发生就激活线程。接收事件使用下面的函数接口：

```
rt_err_t rt_event_recv(rt_event_t event,
                      rt_uint32_t set,
                      rt_uint8_t option,
                      rt_int32_t timeout,
                      rt_uint32_t* recved);
```

当用户调用这个接口时，系统首先根据 `set` 参数和接收选项 `option` 来判断它要接收的事件是否发生，如果已经发生，则根据参数 `option` 上是否设置有 `RT_EVENT_FLAG_CLEAR` 来决定是否重置事件的相应标志位，然后返回（其中 `recved` 参数返回接收到的事件）；如果没有发生，则把等待的 `set` 和 `option` 参数填入线程本身的结构中，然后把线程挂起在此事件上，直到其等待的事件满足条件或等待时间超过指定的超时时间。如果超时时间设置为零，则表示当线程要接受的事件没有满足其要求时就不等待，而直接返回 `-RTETIMEOUT`。下表描述了该函数的输入参数与返回值：

`rt_event_recv()` 的输入参数和返回值

参数	描述
<code>event</code>	事件集对象的句柄
<code>set</code>	接收线程感兴趣的事件
<code>option</code>	接收选项

参数	描述
timeout	指定超时时间
recvied	指向接收到的事件
返回	—
RT_EOK	成功
-RTETIMEOUT	超时
-RT_ERROR	错误

option 的值可取:

```
/* 选择 逻辑与 或 逻辑或 的方式接收事件 */
RT_EVENT_FLAG_OR
RT_EVENT_FLAG_AND

/* 选择清除重置事件标志位 */
RT_EVENT_FLAG_CLEAR
```

5.3.4 事件集应用示例

这是事件集的应用例程，例子中初始化了一个事件集，两个线程。一个线程等待自己关心的事件发生，另外一个线程发送事件，如代码清单 6-5 例所示：

事件集的使用例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      9
#define THREAD_TIMESLICE     5

#define EVENT_FLAG3 (1 << 3)
#define EVENT_FLAG5 (1 << 5)

/* 事件控制块 */
static struct rt_event event;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口函数 */
static void thread1_recv_event(void *param)
{
    rt_uint32_t e;
```

```
/* 第一次接收事件，事件 3 或事件 5 任意一个可以触发线程 1，接收完后清除事件标志 */
if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                  RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                  RT_WAITING_FOREVER, &e) == RT_EOK)
{
    rt_kprintf("thread1: OR recv event 0x%x\n", e);
}

rt_kprintf("thread1: delay 1s to prepare the second event\n");
rt_thread_mdelay(1000);

/* 第二次接收事件，事件 3 和事件 5 均发生时才可以触发线程 1，接收完后清除事件标志 */
if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                  RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                  RT_WAITING_FOREVER, &e) == RT_EOK)
{
    rt_kprintf("thread1: AND recv event 0x%x\n", e);
}
rt_kprintf("thread1 leave.\n");
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_send_event(void *param)
{
    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event5\n");
    rt_event_send(&event, EVENT_FLAG5);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_kprintf("thread2 leave.\n");
}

int event_sample(void)
{
    rt_err_t result;

    /* 初始化事件对象 */
}
```

```

    result = rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                  "thread1",
                  thread1_recv_event,
                  RT_NULL,
                  &thread1_stack[0],
                  sizeof(thread1_stack),
                  THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                  "thread2",
                  thread2_send_event,
                  RT_NULL,
                  &thread2_stack[0],
                  sizeof(thread2_stack),
                  THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(event_sample, event sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >event_sample
thread2: send event3
thread1: OR recv event 0x8
thread1: delay 1s to prepare the second event
msh >thread2: send event5
thread2: send event3
thread2 leave.
thread1: AND recv event 0x28
thread1 leave.

```

例程演示了事件集的使用方法。线程 1 前后两次接收事件，分别使用了“逻辑或”与“逻辑与”的方法。

5.3.5 事件集的使用场合

事件集可使用于多种场合，它能够在一定程度上替代信号量，用于线程间同步。一个线程或中断服务例程发送一个事件给事件集对象，而后等待的线程被唤醒并对相应的事件进行处理。但是它与信号量不同的是，事件的发送操作在事件未清除前，是不可累计的，而信号量的释放动作是累计的。事件的另一个特性是，接收线程可等待多种事件，即多个事件对应一个线程或多个线程。同时按照线程等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一的释放动作，而不能同时等待多种类型的释放。如下图所示为多事件接收示意图：

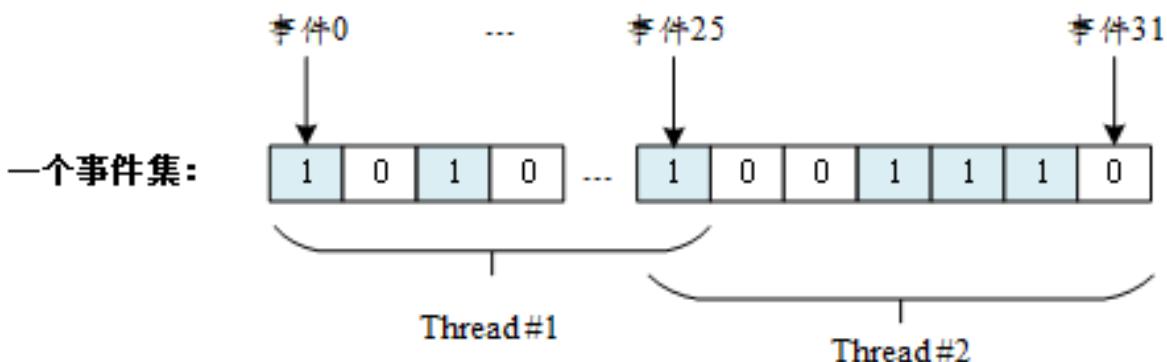


图 5.12: 多事件接收示意图

一个事件集中包含 32 个事件，特定线程只等待、接收它关注的事件。可以是一个线程等待多个事件的到来（线程 1、2 均等待多个事件，事件间可以使用“与”或者“或”逻辑触发线程），也可以是多个线程等待一个事件的到来（事件 25）。当有它们关注的事件发生时，线程将被唤醒并进行后续的处理动作。

第6章

线程间通信

前面一章讲了线程间同步，提到了信号量、互斥量、事件集等概念；本章接着上一章的内容，讲解线程间通信。在裸机编程中，经常会使用全局变量进行功能间的通信，如某些功能可能由于一些操作而改变全局变量的值，另一个功能对此全局变量进行读取，根据读取到的全局变量值执行相应的动作，达到通信协作的目的。**RT-Thread** 中则提供了更多的工具帮助在不同的线程中间传递信息，本章会详细介绍这些工具。学习完本章，大家将学会如何将邮箱、消息队列、信号用于线程间的通信。

6.1 邮箱

邮箱服务是实时操作系统中一种典型的线程间通信方法。举一个简单的例子，有两个线程，线程 1 检测按键状态并发送，线程 2 读取按键状态并根据按键的状态相应地改变 LED 的亮灭。这里就可以使用邮箱的方式进行通信，线程 1 将按键的状态作为邮件发送到邮箱，线程 2 在邮箱中读取邮件获得按键状态并对 LED 执行亮灭操作。

这里的线程 1 也可以扩展为多个线程。例如，共有三个线程，线程 1 检测并发送按键状态，线程 2 检测并发送 ADC 采样信息，线程 3 则根据接收的信息类型不同，执行不同的操作。

6.1.1 邮箱的工作机制

RT-Thread 操作系统的邮箱用于线程间通信，特点是开销比较低，效率较高。邮箱中的每一封邮件只能容纳固定的 4 字节内容（针对 32 位处理系统，指针的大小即为 4 个字节，所以一封邮件恰好能够容纳一个指针）。典型的邮箱也称作交换消息，如下图所示，线程或中断服务例程把一封 4 字节长度的邮件发送到邮箱中，而一个或多个线程可以从邮箱中接收这些邮件并进行处理。



图 6.1: 邮箱工作示意图

非阻塞方式的邮件发送过程能够安全的应用于中断服务中，是线程、中断服务、定时器向线程发送消息的有效手段。通常来说，邮件收取过程可能是阻塞的，这取决于邮箱中是否有邮件，以及收取邮件时设置的超时时间。当邮箱中不存在邮件且超时时间不为 0 时，邮件收取过程将变成阻塞方式。在这类情况下，只能由线程进行邮件的收取。

当一个线程向邮箱发送邮件时，如果邮箱没满，将把邮件复制到邮箱中。如果邮箱已经满了，发送线程可以设置超时时间，选择等待挂起或直接返回 - RT_EFULL。如果发送线程选择挂起等待，那么当邮箱中的邮件被收取而空出空间来时，等待挂起的发送线程将被唤醒继续发送。

当一个线程从邮箱中接收邮件时，如果邮箱是空的，接收线程可以选择是否等待挂起直到收到新的邮件而唤醒，或可以设置超时时间。当达到设置的超时时间，邮箱依然未收到邮件时，这个选择超时等待的线程将被唤醒并返回 - RTETIMEOUT。如果邮箱中存在邮件，那么接收线程将复制邮箱中的 4 个字节邮件到接收缓存中。

6.1.2 邮箱控制块

在 RT-Thread 中，邮箱控制块是操作系统用于管理邮箱的一个数据结构，由结构体 struct rt_mailbox 表示。另外一种 C 表达方式 rt_mailbox_t，表示的是邮箱的句柄，在 C 语言中的实现是邮箱控制块的指针。邮箱控制块结构的详细定义请见以下代码：

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;           /* 邮箱缓冲区的开始地址 */
    rt_uint16_t size;               /* 邮箱缓冲区的大小 */

    rt_uint16_t entry;              /* 邮箱中邮件的数目 */
    rt_uint16_t in_offset, out_offset; /* 邮箱缓冲的进出指针 */
    rt_list_t suspend_sender_thread; /* 发送线程的挂起等待队列 */
};

typedef struct rt_mailbox* rt_mailbox_t;
```

rt_mailbox 对象从 rt_ipc_object 中派生，由 IPC 容器所管理。

6.1.3 邮箱的管理方式

邮箱控制块是一个结构体，其中含有事件相关的重要参数，在邮箱的功能实现中起重要的作用。邮箱的相关接口如下图所示，对一个邮箱的操作包含：创建 / 初始化邮箱、发送邮件、接收邮件、删除 / 脱离邮箱。

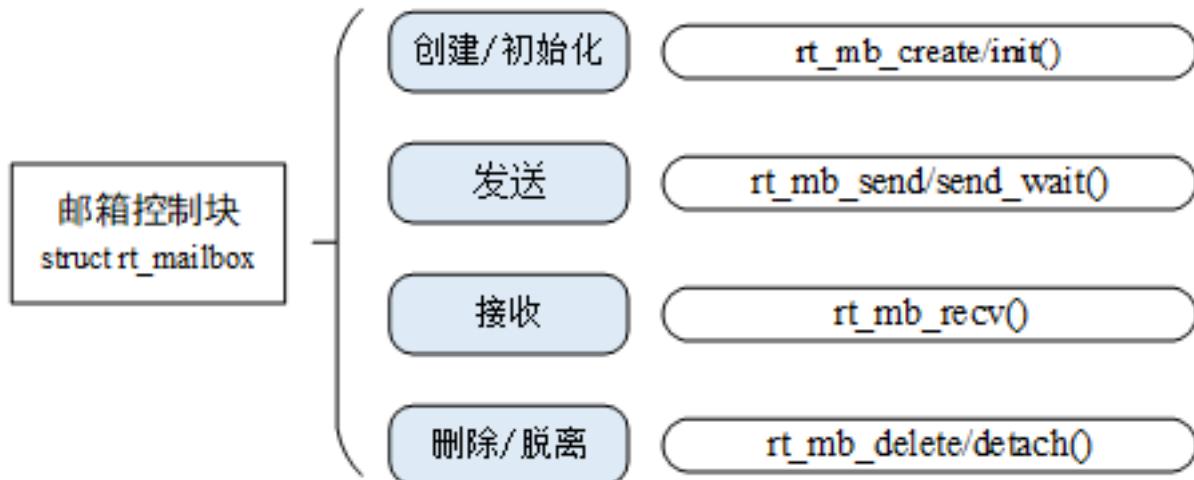


图 6.2: 邮箱相关接口

6.1.3.1 创建和删除邮箱

动态创建一个邮箱对象可以调用如下的函数接口：

```
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8_t flag);
```

创建邮箱对象时会先从对象管理器中分配一个邮箱对象，然后给邮箱动态分配一块内存空间用来存放邮件，这块内存的大小等于邮件大小（4字节）与邮箱容量的乘积，接着初始化接收邮件数目和发送邮件在邮箱中的偏移量。下表描述了该函数的输入参数与返回值：

`rt_mb_create()` 的输入参数和返回值

参数	描述
name	邮箱名称
size	邮箱容量
flag	邮箱标志，它可以取如下数值： RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
RT_NULL	创建失败
邮箱对象的句柄	创建成功

当用 `rt_mb_create()` 创建的邮箱不再被使用时，应该删除它来释放相应的系统资源，一旦操作完成，邮箱将被永久性的删除。删除邮箱的函数接口如下：

```
rt_err_t rt_mb_delete (rt_mailbox_t mb);
```

删除邮箱时，如果有线程被挂起在该邮箱对象上，内核先唤醒挂起在该邮箱上的所有线程（线程返回值是 - RT_ERROR），然后再释放邮箱使用的内存，最后删除邮箱对象。下表描述了该函数的输入参数与返回值：

`rt_mb_delete()` 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
返回	—
RT_EOK	成功

6.1.3.2 初始化和脱离邮箱

初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态邮箱对象的初始化。与创建邮箱不同的是，静态邮箱对象的内存是在系统编译时由编译器分配的，一般放于读写数据段或未初始化数据段中，其余的初始化工作与创建邮箱时相同。函数接口如下：

```
rt_err_t rt_mb_init(rt_mailbox_t mb,
                     const char* name,
                     void* msgpool,
                     rt_size_t size,
                     rt_uint8_t flag)
```

初始化邮箱时，该函数接口需要获得用户已经申请获得的邮箱对象控制块，缓冲区的指针，以及邮箱名称和邮箱容量（能够存储的邮件数）。下表描述了该函数的输入参数与返回值：

rt_mb_init() 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
name	邮箱名称
msgpool	缓冲区指针
size	邮箱容量
flag	邮箱标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
RT_EOK	成功

这里的 size 参数指定的是邮箱的容量，即如果 msgpool 指向的缓冲区的字节数是 N，那么邮箱容量应该是 N/4。

脱离邮箱将把静态初始化的邮箱对象从内核对象管理器中脱离。脱离邮箱使用下面的接口：

```
rt_err_t rt_mb_detach(rt_mailbox_t mb);
```

使用该函数接口后，内核先唤醒所有挂在该邮箱上的线程（线程获得返回值是 - RT_ERROR），然后将该邮箱对象从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

rt_mb_detach() 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
返回	—
RT_EOK	成功

6.1.3.3 发送邮件

线程或者中断服务程序可以通过邮箱给其他线程发送邮件，发送邮件函数接口如下：

```
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32_t value);
```

发送的邮件可以是 32 位任意格式的数据，一个整型值或者一个指向缓冲区的指针。当邮箱中的邮件已经满时，发送邮件的线程或者中断程序会收到 -RT_EFULL 的返回值。下表描述了该函数的输入参数与返回值：

rt_mb_send() 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
value	邮件内容
返回	—
RT_EOK	发送成功
-RT_EFULL	邮箱已经满了

6.1.3.4 等待方式发送邮件

用户也可以通过如下的函数接口向指定邮箱发送邮件：

```
rt_err_t rt_mb_send_wait (rt_mailbox_t mb,
                           rt_uint32_t value,
                           rt_int32_t timeout);
```

rt_mb_send_wait() 与 rt_mb_send() 的区别在于有等待时间，如果邮箱已经满了，那么发送线程将根据设定的 timeout 参数等待邮箱中因为收取邮件而空出空间。如果设置的超时时间到达依然没有空出空间，这时发送线程将被唤醒并返回错误码。下表描述了该函数的输入参数与返回值：

rt_mb_send_wait() 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
value	邮件内容

参数	描述
timeout	超时时间
返回	——
RT_EOK	发送成功
-RTETIMEOUT	超时
-RT_ERROR	失败, 返回错误

6.1.3.5 接收邮件

只有当接收者接收的邮箱中有邮件时，接收者才能立即取到邮件并返回 RT_EOK 的返回值，否则接收线程会根据超时时间设置，或挂起在邮箱的等待线程队列上，或直接返回。接收邮件函数接口如下：

```
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32_t* value, rt_int32_t timeout);
```

接收邮件时，接收者需指定接收邮件的邮箱句柄，并指定接收到的邮件存放位置以及最多能够等待的超时时间。如果接收时设定了超时，当指定的时间内依然未收到邮件时，将返回 -RTETIMEOUT。下表描述了该函数的输入参数与返回值：

rt_mb_recv() 的输入参数和返回值

参数	描述
mb	邮箱对象的句柄
value	邮件内容
timeout	超时时间
返回	——
RT_EOK	发送成功
-RTETIMEOUT	超时
-RT_ERROR	失败, 返回错误

6.1.4 邮箱使用示例

这是一个邮箱的应用例程，初始化 2 个静态线程，一个静态的邮箱对象，其中一个线程往邮箱中发送邮件，一个线程往邮箱中收取邮件。如下代码所示：

邮箱的使用例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      10
#define THREAD_TIMESLICE     5
```

```
/* 邮箱控制块 */
static struct rt_mailbox mb;
/* 用于放邮件的内存池 */
static char mb_pool[128];

static char mb_str1[] = "I'm a mail!";
static char mb_str2[] = "this is another mail!";
static char mb_str3[] = "over";

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    char *str;

    while (1)
    {
        rt_kprintf("thread1: try to recv a mail\n");

        /* 从邮箱中收取邮件 */
        if (rt_mb_recv(&mb, (rt_uint32_t *)&str, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: get a mail from mailbox, the content:%s\n", str);
            if (str == mb_str3)
                break;

            /* 延时 100ms */
            rt_thread_mdelay(100);
        }
    }
    /* 执行邮箱对象脱离 */
    rt_mb_detach(&mb);
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_uint8_t count;

    count = 0;
    while (count < 10)
    {
```

```
count++;
if (count & 0x1)
{
    /* 发送 mb_str1 地址到邮箱中 */
    rt_mb_send(&mb, (rt_uint32_t)&mb_str1);
}
else
{
    /* 发送 mb_str2 地址到邮箱中 */
    rt_mb_send(&mb, (rt_uint32_t)&mb_str2);
}

/* 延时 200ms */
rt_thread_mdelay(200);
}

/* 发送邮件告诉线程 1, 线程 2 已经运行结束 */
rt_mb_send(&mb, (rt_uint32_t)&mb_str3);
}

int mailbox_sample(void)
{
    rt_err_t result;

    /* 初始化一个 mailbox */
    result = rt_mb_init(&mb,
                        "mbt",
                        &mb_pool[0],
                        sizeof(mb_pool) / 4,
                        /* 件占 4 字节 */
                        RT_IPC_FLAG_FIFO);
    /* 名称是 mbt */
    /* 邮箱用到的内存池是 mb_pool */
    /* 邮箱中的邮件数目, 因为一封邮
       件占 4 字节 */
    /* 采用 FIFO 方式进行线程等待 */

    if (result != RT_EOK)
    {
        rt_kprintf("init mailbox failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                  "thread1",
                  thread1_entry,
                  RT_NULL,
                  &thread1_stack[0],
                  sizeof(thread1_stack),
                  THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                  "thread2",
                  thread2_entry,
```

```

        RT_NULL,
        &thread2_stack[0],
        sizeof(thread2_stack),
        THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(mailbox_sample, mailbox sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >mailbox_sample
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
msh >thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
...
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:over

```

例程演示了邮箱的使用方法。线程 2 发送邮件，共发送 11 次；线程 1 接收邮件，共接收到 11 封邮件，将邮件内容打印出来，并判断结束。

6.1.5 邮箱的使用场合

邮箱是一种简单的线程间消息传递方式，特点是开销比较低，效率较高。在 RT-Thread 操作系统的实现中能够一次传递一个 4 字节大小的邮件，并且邮箱具备一定的存储功能，能够缓存一定数量的邮件数（邮件数由创建、初始化邮箱时指定的容量决定）。邮箱中一封邮件的最大长度是 4 字节，所以邮箱能够用于不超过 4 字节的消息传递。由于在 32 系统上 4 字节的内容恰好可以放置一个指针，因此当需要在线程间传递比较大的消息时，可以把指向一个缓冲区的指针作为邮件发送到邮箱中，即邮箱也可以传递指针，例如：

```

struct msg
{
    rt_uint8_t *data_ptr;
    rt_uint32_t data_size;
};

```

对于这样一个消息结构体，其中包含了指向数据的指针 `data_ptr` 和数据块长度的变量 `data_size`。当一个线程需要把这个消息发送给另外一个线程时，可以采用如下的操作：

```

struct msg* msg_ptr;

msg_ptr = (struct msg*)rt_malloc(sizeof(struct msg));
msg_ptr->data_ptr = ...; /* 指向相应的数据块地址 */
msg_ptr->data_size = len; /* 数据块的长度 */
/* 发送这个消息指针给 mb 邮箱 */
rt_mb_send(mb, (rt_uint32_t)msg_ptr);

```

而在接收线程中，因为收取过来的是指针，而 `msg_ptr` 是一个新分配出来的内存块，所以在接收线程处理完毕后，需要释放相应的内存块：

```

struct msg* msg_ptr;
if (rt_mb_recv(mb, (rt_uint32_t*)&msg_ptr) == RT_EOK)
{
    /* 在接收线程处理完毕后，需要释放相应的内存块 */
    rt_free(msg_ptr);
}

```

6.2 消息队列

消息队列是另一种常用的线程间通讯方式，是邮箱的扩展。可以应用在多种场合：线程间的消息交换、使用串口接收不定长数据等。

6.2.1 消息队列的工作机制

消息队列能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

如下图所示，线程或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个线程也可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常将先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则 (FIFO)。

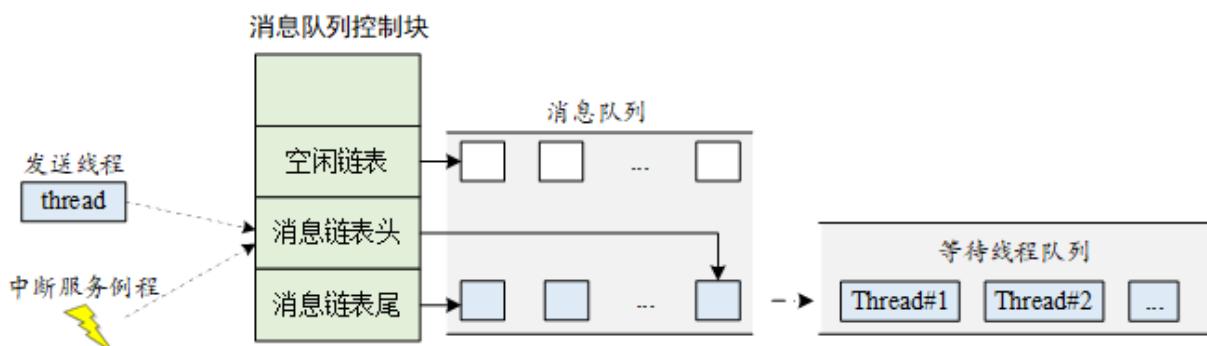


图 6.3: 消息队列工作示意图

RT-Thread 操作系统的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：消息队列名称、内存缓冲区、消息大小以及队列长度等。同时每个消息队列对象中包含着多个消

息框，每个消息框可以存放一条消息；消息队列中的第一个和最后一个消息框被分别称为消息链表头和消息链表尾，对应于消息队列控制块中的 `msg_queue_head` 和 `msg_queue_tail`；有些消息框可能是空的，它们通过 `msg_queue_free` 形成一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

6.2.2 消息队列控制块

在 RT-Thread 中，消息队列控制块是操作系统用于管理消息队列的一个数据结构，由结构体 `struct rt_messagequeue` 表示。另外一种 C 表达方式 `rt_mq_t`，表示的是消息队列的句柄，在 C 语言中的实现是消息队列控制块的指针。消息队列控制块结构的详细定义请见以下代码：

```
struct rt_messagequeue
{
    struct rt_ipc_object parent;

    void* msg_pool; /* 指向存放消息的缓冲区的指针 */

    rt_uint16_t msg_size; /* 每个消息的长度 */
    rt_uint16_t max_msgs; /* 最大能够容纳的消息数 */

    rt_uint16_t entry; /* 队列中已有的消息数 */

    void* msg_queue_head; /* 消息链表头 */
    void* msg_queue_tail; /* 消息链表尾 */
    void* msg_queue_free; /* 空闲消息链表 */

    rt_list_t suspend_sender_thread; /* 发送线程的挂起等待队列 */
};

typedef struct rt_messagequeue* rt_mq_t;
```

`rt_messagequeue` 对象从 `rt_ipc_object` 中派生，由 IPC 容器所管理。

6.2.3 消息队列的管理方式

消息队列控制块是一个结构体，其中含有消息队列相关的重要参数，在消息队列的功能实现中起重要的作用。消息队列的相关接口如下图所示，对一个消息队列的操作包含：创建消息队列 - 发送消息 - 接收消息 - 删除消息队列。

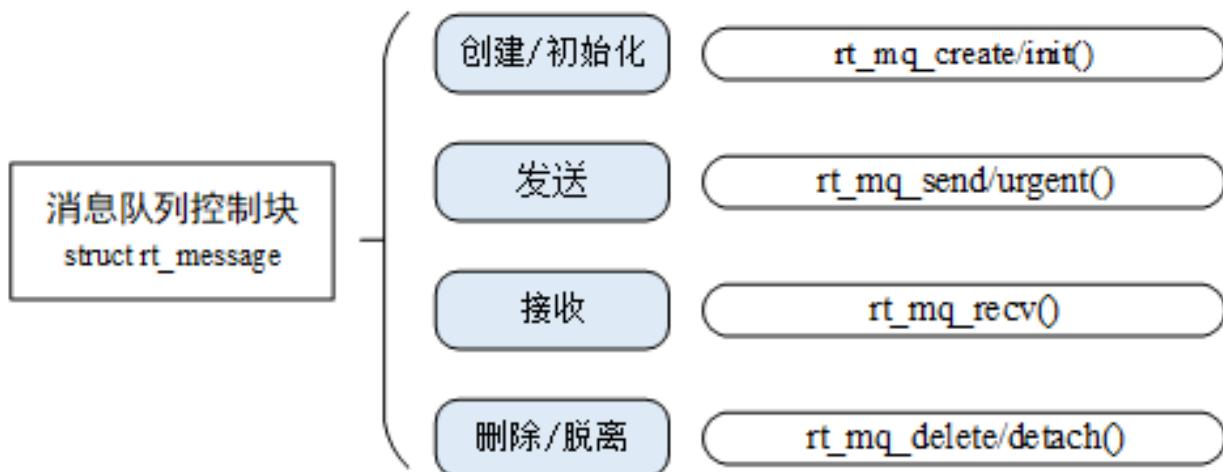


图 6.4: 消息队列相关接口

6.2.3.1 创建和删除消息队列

消息队列在使用前，应该被创建出来，或对已有的静态消息队列对象进行初始化，创建消息队列的函数接口如下所示：

```
rt_mq_t rt_mq_create(const char* name, rt_size_t msg_size,
                      rt_size_t max_msgs, rt_uint8_t flag);
```

创建消息队列时先从对象管理器中分配一个消息队列对象，然后给消息队列对象分配一块内存空间，组织成空闲消息链表，这块内存的大小 = [消息大小 + 消息头（用于链表连接）的大小] × 消息队列最大个数，接着再初始化消息队列，此时消息队列为空。下表描述了该函数的输入参数与返回值：

`rt_mq_create()` 的输入参数和返回值

参数	描述
<code>name</code>	消息队列的名称
<code>msg_size</code>	消息队列中一条消息的最大长度，单位字节
<code>max_msgs</code>	消息队列的最大个数
<code>flag</code>	消息队列采用的等待方式，它可以取如下数值： <code>RT_IPC_FLAG_FIFO</code> 或 <code>RT_IPC_FLAG_PRIO</code>
返回	—
<code>RT_EOK</code>	发送成功
消息队列对象的句柄	成功
<code>RT_NULL</code>	失败

当消息队列不再被使用时，应该删除它以释放系统资源，一旦操作完成，消息队列将被永久性地删除。删除消息队列的函数接口如下：

```
rt_err_t rt_mq_delete(rt_mq_t mq);
```

删除消息队列时，如果有线程被挂起在该消息队列等待队列上，则内核先唤醒挂起在该消息等待队列上的所有线程（线程返回值是 - RT_ERROR），然后再释放消息队列使用的内存，最后删除消息队列对象。下表描述了该函数的输入参数与返回值：

`rt_mq_delete()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
返回	—
RT_EOK	成功

6.2.3.2 初始化和脱离消息队列

初始化静态消息队列对象跟创建消息队列对象类似，只是静态消息队列对象的内存是在系统编译时由编译器分配的，一般放于读数据段或未初始化数据段中。在使用这类静态消息队列对象前，需要进行初始化。初始化消息队列对象的函数接口如下：

```
rt_err_t rt_mq_init(rt_mq_t mq, const char* name,
                     void *msgpool, rt_size_t msg_size,
                     rt_size_t pool_size, rt_uint8_t flag);
```

初始化消息队列时，该接口需要用户已经申请获得的消息队列对象的句柄（即指向消息队列对象控制块的指针）、消息队列名、消息缓冲区指针、消息大小以及消息队列缓冲区大小。如下图所示，消息队列初始化后所有消息都挂在空闲消息链表上，消息队列为空。下表描述了该函数的输入参数与返回值：

`rt_mq_init()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
name	消息队列的名称
msgpool	指向存放消息的缓冲区的指针
msg_size	消息队列中一条消息的最大长度，单位字节
pool_size	存放消息的缓冲区大小
flag	消息队列采用的等待方式，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	—
RT_EOK	成功

脱离消息队列将使消息队列对象被从内核对象管理器中脱离。脱离消息队列使用下面的接口：

```
rt_err_t rt_mq_detach(rt_mq_t mq);
```

使用该函数接口后，内核先唤醒所有挂在该消息等待队列对象上的线程（线程返回值是 -RT_ERROR），然后将该消息队列对象从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

`rt_mq_detach()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
返回	—
RT_EOK	成功

6.2.3.3 发送消息

线程或者中断服务程序都可以给消息队列发送消息。当发送消息时，消息队列对象先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。当且仅当空闲消息链表上有可用的空闲消息块时，发送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的线程或者中断程序会收到一个错误码 (-RT_EFULL)。发送消息的函数接口如下：

```
rt_err_t rt_mq_send (rt_mq_t mq, void* buffer, rt_size_t size);
```

发送消息时，发送者需指定发送的消息队列的对象句柄（即指向消息队列控制块的指针），并且指定发送的消息内容以及消息大小。如下图所示，在发送一个普通消息之后，空闲消息链表上的队首消息被转移到了消息队列尾。下表描述了该函数的输入参数与返回值：

`rt_mq_send()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
buffer	消息内容
size	消息大小
返回	—
RT_EOK	成功
-RT_EFULL	消息队列已满
-RT_ERROR	失败，表示发送的消息长度大于消息队列中消息的最大长度

6.2.3.4 等待方式发送消息

用户也可以通过如下的函数接口向指定的消息队列中发送消息：

```
rt_err_t rt_mq_send_wait(rt_mq_t      mq,
```

```
const void *buffer,
rt_size_t size,
rt_int32_t timeout);
```

`rt_mq_send_wait()` 与 `rt_mq_send()` 的区别在于有等待时间，如果消息队列已经满了，那么发送线程将根据设定的 `timeout` 参数进行等待。如果设置的超时时间到达依然没有空出空间，这时发送线程将被唤醒并返回错误码。下表描述了该函数的输入参数与返回值：

`rt_mq_send_wait()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
buffer	消息内容
size	消息大小
timeout	超时时间
返回	—
RT_EOK	成功
-RT_EFULL	消息队列已满
-RT_ERROR	失败，表示发送的消息长度大于消息队列中消息的最大长度

6.2.3.5 发送紧急消息

发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，从空闲消息链表上取下来的消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者就能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的函数接口如下：

```
rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size);
```

下表描述了该函数的输入参数与返回值：

`rt_mq_urgent()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
buffer	消息内容
size	消息大小
返回	—
RT_EOK	成功
-RT_EFULL	消息队列已满
-RT_ERROR	失败

6.2.3.6 接收消息

当消息队列中有消息时，接收者才能接收消息，否则接收者会根据超时时间设置，或挂起在消息队列的等待线程队列上，或直接返回。接收消息函数接口如下：

```
rt_err_t rt_mq_recv (rt_mq_t mq, void* buffer,
                      rt_size_t size, rt_int32_t timeout);
```

接收消息时，接收者需指定存储消息的消息队列对象句柄，并且指定一个内存缓冲区，接收到的消息内容将被复制到该缓冲区里。此外，还需指定未能及时取到消息时的超时时间。如下图所示，接收一个消息后消息队列上的队首消息被转移到了空闲消息链表的尾部。下表描述了该函数的输入参数与返回值：

`rt_mq_recv()` 的输入参数和返回值

参数	描述
mq	消息队列对象的句柄
buffer	消息内容
size	消息大小
timeout	指定的超时时间
返回	—
RT_EOK	成功收到
-RTETIMEOUT	超时
-RT_ERROR	失败，返回错误

6.2.4 消息队列应用示例

这是一个消息队列的应用例程，例程中初始化了 2 个静态线程，一个线程会从消息队列中收取消息；另一个线程会定时给消息队列发送普通消息和紧急消息，如下代码所示：

消息队列的使用例程

```
#include <rtthread.h>

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static rt_uint8_t msg_pool[2048];

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
/* 线程 1 入口函数 */
static void thread1_entry(void *parameter)
{
    char buf = 0;
```

```
rt_uint8_t cnt = 0;

while (1)
{
    /* 从消息队列中接收消息 */
    if (rt_mq_recv(&mq, &buf, sizeof(buf), RT_WAITING_FOREVER) == RT_EOK)
    {
        rt_kprintf("thread1: recv msg from msg queue, the content:%c\n", buf);
        if (cnt == 19)
        {
            break;
        }
    }
    /* 延时 50ms */
    cnt++;
    rt_thread_mdelay(50);
}
rt_kprintf("thread1: detach mq \n");
rt_mq_detach(&mq);
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    int result;
    char buf = 'A';
    rt_uint8_t cnt = 0;

    while (1)
    {
        if (cnt == 8)
        {
            /* 发送紧急消息到消息队列中 */
            result = rt_mq_urgent(&mq, &buf, 1);
            if (result != RT_EOK)
            {
                rt_kprintf("rt_mq_urgent ERR\n");
            }
            else
            {
                rt_kprintf("thread2: send urgent message - %c\n", buf);
            }
        }
        else if (cnt>= 20)/* 发送 20 次消息之后退出 */
        {
            rt_kprintf("message queue stop send, thread2 quit\n");
        }
    }
}
```

```
        break;
    }
    else
    {
        /* 发送消息到消息队列中 */
        result = rt_mq_send(&mq, &buf, 1);
        if (result != RT_EOK)
        {
            rt_kprintf("rt_mq_send ERR\n");
        }

        rt_kprintf("thread2: send message - %c\n", buf);
    }
    buf++;
    cnt++;
    /* 延时 5ms */
    rt_thread_mdelay(5);
}

/* 消息队列示例的初始化 */
int msgq_sample(void)
{
    rt_err_t result;

    /* 初始化消息队列 */
    result = rt_mq_init(&mq,
                        "mqt",
                        &msg_pool[0],           /* 内存池指向 msg_pool */
                        1,                      /* 每个消息的大小是 1 字节 */
                        sizeof(msg_pool),       /* 内存池的大小是 msg_pool 的大小
                                              */
                        RT_IPC_FLAG_FIFO);     /* 如果有多个线程等待，按照先来先得
                                              到的方法分配消息 */

    if (result != RT_EOK)
    {
        rt_kprintf("init message queue failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                  "thread1",
                  thread1_entry,
                  RT_NULL,
                  &thread1_stack[0],
                  sizeof(thread1_stack), 25, 5);
    rt_thread_startup(&thread1);
```

```
rt_thread_init(&thread2,
               "thread2",
               thread2_entry,
               RT_NULL,
               &thread2_stack[0],
               sizeof(thread2_stack), 25, 5);
rt_thread_startup(&thread2);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(msgq_sample, msgq sample);
```

仿真运行结果如下：

```
\ | /
- RT -      Thread Operating System
 / | \    3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >msgq_sample
msh >thread2: send message - A
thread1: recv msg from msg queue, the content:A
thread2: send message - B
thread2: send message - C
thread2: send message - D
thread2: send message - E
thread1: recv msg from msg queue, the content:B
thread2: send message - F
thread2: send message - G
thread2: send message - H
thread2: send urgent message - I
thread2: send message - J
thread1: recv msg from msg queue, the content:I
thread2: send message - K
thread2: send message - L
thread2: send message - M
thread2: send message - N
thread2: send message - O
thread1: recv msg from msg queue, the content:C
thread2: send message - P
thread2: send message - Q
thread2: send message - R
thread2: send message - S
thread2: send message - T
thread1: recv msg from msg queue, the content:D
message queue stop send, thread2 quit
thread1: recv msg from msg queue, the content:E
thread1: recv msg from msg queue, the content:F
```

```
thread1: recv msg from msg queue, the content:G
...
thread1: recv msg from msg queue, the content:T
thread1: detach mq
```

例程演示了消息队列的使用方法。线程 1 会从消息队列中收取消息；线程 2 定时给消息队列发送普通消息和紧急消息。由于线程 2 发送消息 “I” 是紧急消息，会直接插入消息队列的队首，所以线程 1 在接收到消息 “B” 后，接收的是该紧急消息，之后才接收消息 “C”。

6.2.5 消息队列的使用场合

消息队列可以应用于发送不定长消息的场合，包括线程与线程间的消息交换，以及中断服务例程中给线程发送消息（中断服务例程不能接收消息）。下面分发送消息和同步消息两部分来介绍消息队列的使用。

6.2.5.1 发送消息

消息队列和邮箱的明显不同是消息的长度并不限定在 4 个字节以内；另外，消息队列也包括了一个发送紧急消息的函数接口。但是当创建的是一个所有消息的最大长度是 4 字节的消息队列时，消息队列对象将蜕化成邮箱。这个不限定长度的消息，也及时的反应到了代码编写的场上，同样是类似邮箱的代码：

```
struct msg
{
    rt_uint8_t *data_ptr; /* 数据块首地址 */
    rt_uint32_t data_size; /* 数据块大小 */
};
```

和邮箱例子相同的消息结构定义，假设依然需要发送这样一个消息给接收线程。在邮箱例子中，这个结构只能够发送指向这个结构的指针（在函数指针被发送过去后，接收线程能够正确的访问指向这个地址的内容，通常这块数据需要留给接收线程来释放）。而使用消息队列的方式则大不相同：

```
void send_op(void *data, rt_size_t length)
{
    struct msg msg_ptr;

    msg_ptr.data_ptr = data; /* 指向相应的数据块地址 */
    msg_ptr.data_size = length; /* 数据块的长度 */

    /* 发送这个消息指针给 mq 消息队列 */
    rt_mq_send(mq, (void*)&msg_ptr, sizeof(struct msg));
}
```

注意，上面的代码中，是把一个局部变量的数据内容发送到了消息队列中。在接收线程中，同样也采用局部变量进行消息接收的结构体：

```
void message_handler()
{
    struct msg msg_ptr; /* 用于放置消息的局部变量 */
```

```
/* 从消息队列中接收消息到 msg_ptr 中 */
if (rt_mq_recv(mq, (void*)&msg_ptr, sizeof(struct msg)) == RT_EOK)
{
    /* 成功接收到消息，进行相应的数据处理 */
}
}
```

因为消息队列是直接的数据内容复制，所以在上面的例子中，都采用了局部变量的方式保存消息结构体，这样也就免去动态内存分配的烦恼了（也就不用担心，接收线程在接收到消息时，消息内存空间已经被释放）。

6.2.5.2 同步消息

在一般的系统设计中会经常遇到要发送同步消息的问题，这个时候就可以根据当时状态的不同选择相应的实现：两个线程间可以采用【消息队列 + 信号量或邮箱】的形式实现。发送线程通过消息发送的形式发送相应的消息给消息队列，发送完毕后希望获得接收线程的收到确认，工作示意图如下图所示：



图 6.5: 同步消息示意图

根据消息确认的不同，可以把消息结构体定义成：

```
struct msg
{
    /* 消息结构其他成员 */
    struct rt_mailbox ack;
};

/* 或者 */

struct msg
{
    /* 消息结构其他成员 */
    struct rt_semaphore ack;
};
```

第一种类型的消息使用了邮箱来作为确认标志，而第二种类型的消息采用了信号量来作为确认标志。邮箱作为确认标志，代表着接收线程能够通知一些状态值给发送线程；而信号量作为确认标志只能够单一的通知发送线程，消息已经确认接收。

6.3 信号

信号（又称为软中断信号），在软件层次上是对中断机制的一种模拟，在原理上，一个线程收到一个信号与处理器收到一个中断请求可以说是类似的。

6.3.1 信号的工作机制

信号在 RT-Thread 中用作异步通信，POSIX 标准定义了 `sigset_t` 类型来定义一个信号集，然而 `sigset_t` 类型在不同的系统可能有不同的定义方式，在 RT-Thread 中，将 `sigset_t` 定义成了 `unsigned long` 型，并命名为 `rt_sigset_t`，应用程序能够使用的信号为 `SIGUSR1` (10) 和 `SIGUSR2` (12)。

信号本质是软中断，用来通知线程发生了异步事件，用做线程之间的异常通知、应急处理。一个线程不必通过任何操作来等待信号的到达，事实上，线程也不知道信号到底什么时候到达，线程之间可以互相通过调用 `rt_thread_kill()` 发送软中断信号。

收到信号的线程对各种信号有不同的处理方法，处理方法可以分为三类：

第一种是类似中断的处理程序，对于需要处理的信号，线程可以指定处理函数，由该函数来处理。

第二种方法是，忽略某个信号，对该信号不做任何处理，就像未发生过一样。

第三种方法是，对该信号的处理保留系统的默认值。

如下图所示，假设线程 1 需要对信号进行处理，首先线程 1 安装一个信号并解除阻塞，并在安装的同时设定了对信号的异常处理方式；然后其他线程可以给线程 1 发送信号，触发线程 1 对该信号的处理。

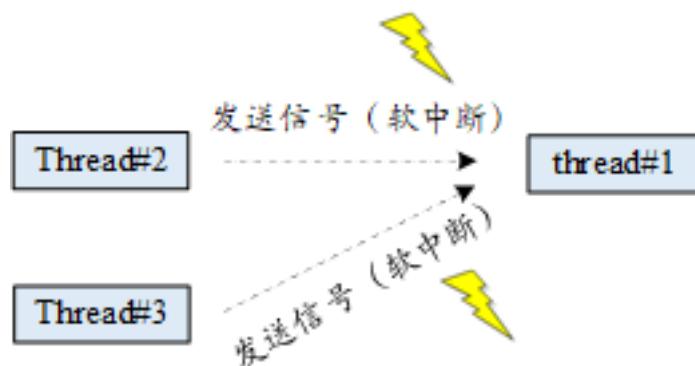


图 6.6: 信号工作机制

当信号被传递给线程 1 时，如果它正处于挂起状态，那会把状态改为就绪状态去处理对应的信号。如果它正处于运行状态，那么会在它当前的线程栈基础上建立新栈帧空间去处理对应的信号，需要注意的是使用的线程栈大小也会相应增加。

6.3.2 信号的管理方式

对于信号的操作，有以下几种：安装信号、阻塞信号、阻塞解除、信号发送、信号等待。信号的接口详见下图：



图 6.7: 信号相关接口

6.3.2.1 安装信号

如果线程要处理某一信号，那么就要在线程中安装该信号。安装信号主要用来确定信号值及线程针对该信号值的动作之间的映射关系，即线程将要处理哪个信号，该信号被传递给线程时，将执行何种操作。详细定义请见以下代码：

```
rt_sighandler_t rt_signal_install(int signo, rt_sighandler_t[] handler);
```

其中 `rt_sighandler_t` 是定义信号处理函数的函数指针类型。下表描述了该函数的输入参数与返回值：

`rt_signal_install()` 的输入参数和返回值

参数	描述
<code>signo</code>	信号值（只有 <code>SIGUSR1</code> 和 <code>SIGUSR2</code> 是开放给用户使用的，下同）
<code>handler</code>	设置对信号值的处理方式
返回	—
<code>SIG_ERR</code>	错误的信号
安装信号前的 <code>handler</code> 值	成功

在信号安装时设定 `handler` 参数，决定了该信号的不同的处理方法。处理方法可以分为三种：

- 1) 类似中断的处理方式，参数指向当信号发生时用户自定义的处理函数，由该函数来处理。
- 2) 参数设为 `SIG_IGN`，忽略某个信号，对该信号不做任何处理，就像未发生过一样。
- 3) 参数设为 `SIG_DFL`，系统会调用默认的处理函数 `_signal_default_handler()`。

6.3.2.2 阻塞信号

信号阻塞，也可以理解为屏蔽信号。如果该信号被阻塞，则该信号将不会递达给安装此信号的线程，也不会引发软中断处理。调 `rt_signal_mask()` 可以使信号阻塞：

```
void rt_signal_mask(int signo);
```

下表描述了该函数的输入参数:

rt_signal_mask() 函数参数

参数	描述
signo	信号值

6.3.2.3 解除信号阻塞

线程中可以安装好几个信号，使用此函数可以对其中一些信号给予“关注”，那么发送这些信号都会引发该线程的软中断。调用 `rt_signal_unmask()` 可以用来解除信号阻塞:

```
void rt_signal_unmask(int signo);
```

下表描述了该函数的输入参数:

rt_signal_unmask() 函数参数

参数	描述
signo	信号值

6.3.2.4 发送信号

当需要进行异常处理时，可以给设定了处理异常的线程发送信号，调用 `rt_thread_kill()` 可以用来向任何线程发送信号:

```
int rt_thread_kill(rt_thread_t tid, int sig);
```

下表描述了该函数的输入参数与返回值:

rt_thread_kill() 的输入参数和返回值

参数	描述
tid	接收信号的线程
sig	信号值
返回	—
RT_EOK	发送成功
-RT_EINVAL	参数错误

6.3.2.5 等待信号

等待 set 信号的到来，如果没有等到这个信号，则将线程挂起，直到等到这个信号或者等待时间超过指定的超时时间 `timeout`。如果等到了该信号，则将指向该信号体的指针存入 `si`，如下是等待信号的函数。

```
int rt_signal_wait(const rt_sigset_t *set,
                   rt_siginfo_t[] *si, rt_int32_t timeout);
```

其中 `rt_siginfo_t` 是定义信号信息的数据类型，下表描述了该函数的输入参数与返回值：

`rt_signal_wait()` 的输入参数和返回值

参数	描述
<code>set</code>	指定等待的信号
<code>si</code>	指向存储等到信号信息的指针
<code>timeout</code>	指定的等待时间
返回	—
<code>RT_EOK</code>	等到信号
<code>-RTETIMEOUT</code>	超时
<code>-RT EINVAL</code>	参数错误

6.3.3 信号应用示例

这是一个信号的应用例程，如下代码所示。此例程创建了 1 个线程，在安装信号时，信号处理方式设为自定义处理，定义的信号的处理函数为 `thread1_signal_handler()`。待此线程运行起来安装好信号之后，给此线程发送信号。此线程将接收到信号，并打印信息。

信号使用例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

static rt_thread_t tid1 = RT_NULL;

/* 线程 1 的信号处理函数 */
void thread1_signal_handler(int sig)
{
    rt_kprintf("thread1 received signal %d\n", sig);
}

/* 线程 1 的入口函数 */
static void thread1_entry(void *parameter)
{
    int cnt = 0;

    /* 安装信号 */
    rt_signal_install(SIGUSR1, thread1_signal_handler);
```

```
rt_signal_unmask(SIGUSR1);

/* 运行 10 次 */
while (cnt < 10)
{
    /* 线程 1 采用低优先级运行，一直打印计数值 */
    rt_kprintf("thread1 count : %d\n", cnt);

    cnt++;
    rt_thread_mdelay(100);
}

/*
 * 信号示例的初始化
 */
int signal_sample(void)
{
    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                           thread1_entry, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);

    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    rt_thread_mdelay(300);

    /* 发送信号 SIGUSR1 给线程 1 */
    rt_thread_kill(tid1, SIGUSR1);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(signal_sample, signal sample);
```

仿真运行结果如下：

```
\ | /
- RT -      Thread Operating System
 / | \      3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >signal_sample
thread1 count : 0
thread1 count : 1
thread1 count : 2
msh >thread1 received signal 10
thread1 count : 3
thread1 count : 4
```

```
thread1 count : 5  
thread1 count : 6  
thread1 count : 7  
thread1 count : 8  
thread1 count : 9
```

例程中，首先线程安装信号并解除阻塞，然后发送信号给线程。线程接收到信号并打印出了接收到的信号：SIGUSR1（10）。

第 7 章

内存管理

在计算系统中，通常存储空间可以分为两种：内部存储空间和外部存储空间。内部存储空间通常访问速度比较快，能够按照变量地址随机地访问，也就是我们通常所说的 RAM（随机存储器），可以把它理解为电脑的内存；而外部存储空间内所保存的内容相对来说比较固定，即使掉电后数据也不会丢失，这就是通常所讲的 ROM（只读存储器），可以把它理解为电脑的硬盘。

计算机系统中，变量、中间数据一般存放在 RAM 中，只有在实际使用时才将它们从 RAM 调入到 CPU 中进行运算。一些数据需要的内存大小需要在程序运行过程中根据实际情况确定，这就要求系统具有对内存空间进行动态管理的能力，在用户需要一段内存空间时，向系统申请，系统选择一段合适的内存空间分配给用户，用户使用完毕后，再释放回系统，以便系统将该段内存空间回收再利用。

这章主要介绍 RT-Thread 中的两种内存管理方式，分别是动态内存堆管理和静态内存池管理，学完本章，读者会了解 RT-Thread 的内存管理原理及使用方式。

7.1 内存管理的功能特点

由于实时系统中对时间的要求非常严格，内存管理往往要比通用操作系统要求苛刻得多：

1) 分配内存的时间必须是确定的。一般内存管理算法是根据需要存储的数据的长度在内存中去寻找一个与这段数据相适应的空闲内存块，然后将数据存储在里面。而寻找这样一个空闲内存块所耗费的时间是不确定的，因此对于实时系统来说，这就是不可接受的，实时系统必须要保证内存块的分配过程在可预测的确定时间内完成，否则实时任务对外部事件的响应也将变得不可确定。

2) 随着内存不断被分配和释放，整个内存区域会产生越来越多的碎片（因为在使用过程中，申请了一些内存，其中一些释放了，导致内存空间中存在一些小的内存块，它们地址不连续，不能够作为一整块的大内存分配出去），系统中还有足够的空闲内存，但因为它们地址并非连续，不能组成一块连续的完整内存块，会使得程序不能申请到大的内存。对于通用系统而言，这种不恰当的内存分配算法可以通过重新启动系统来解决（每个月或者数个月进行一次），但是对于那些需要常年不间断地工作于野外的嵌入式系统来说，就变得让人无法接受了。

3) 嵌入式系统的资源环境也是不尽相同，有些系统的资源比较紧张，只有数十 KB 的内存可供分配，而有些系统则存在数 MB 的内存，如何为这些不同的系统，选择适合它们的高效率的内存分配算法，就将变得复杂化。

RT-Thread 操作系统在内存管理上，根据上层应用及系统资源的不同，有针对性地提供了不同的内存分配管理算法。总体上可分为两类：内存堆管理与内存池管理，而内存堆管理又根据具体内存设备划分为三种情况：

第一种是针对小内存块的分配管理（小内存管理算法）；

第二种是针对大内存块的分配管理（slab 管理算法）；

第三种是针对多内存堆的分配情况（memheap 管理算法）

7.2 内存堆管理

内存堆管理用于管理一段连续的内存空间，在第三章中介绍过 RT-Thread 的内存分布情况，如下图所示，RT-Thread 将“ZI 段结尾处”到内存尾部的空间用作内存堆。

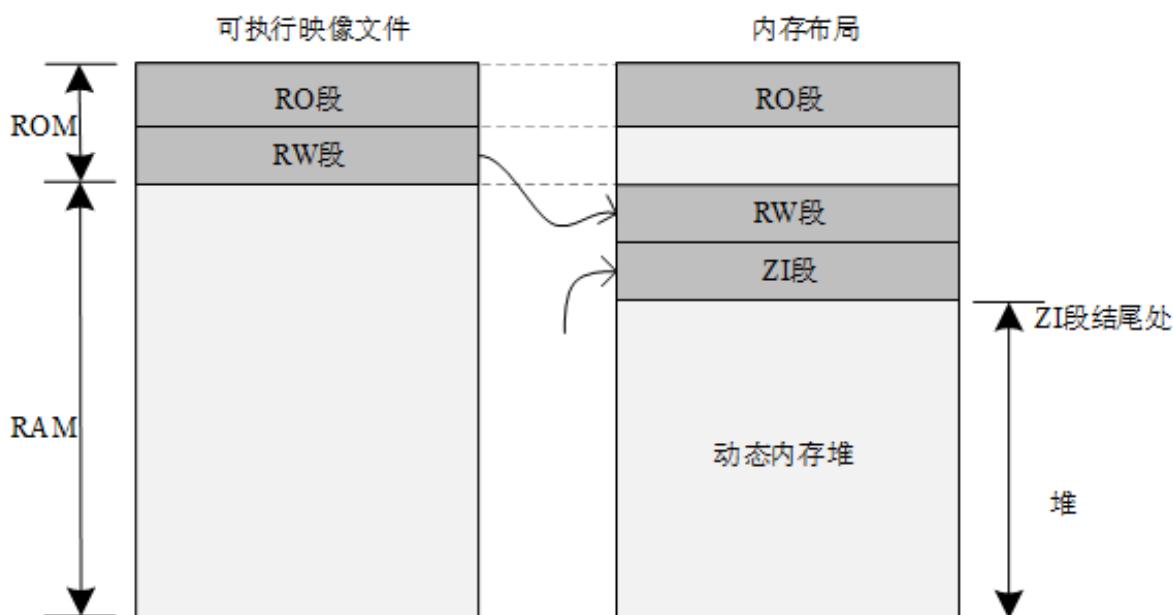


图 7.1: RT-Thread 内存分布

内存堆可以在当前资源满足的情况下，根据用户的需求分配任意大小的内存块。而当用户不需要再使用这些内存块时，又可以释放回堆中供其他应用分配使用。RT-Thread 系统为了满足不同的需求，提供了不同的内存管理算法，分别是小内存管理算法、slab 管理算法和 memheap 管理算法。

小内存管理算法主要针对系统资源比较少，一般用于小于 2MB 内存空间的系统；而 slab 内存管理算法则主要是在系统资源比较丰富时，提供了一种近似多内存池管理算法的快速算法。除上述之外，RT-Thread 还有一种针对多内存堆的管理算法，即 memheap 管理算法。memheap 方法适用于系统存在多个内存堆的情况，它可以将多个内存“粘贴”在一起，形成一个大的内存堆，用户使用起来会非常方便。

这几类内存堆管理算法在系统运行时只能选择其中之一或者完全不使用内存堆管理器，他们提供给应用程序的 API 接口完全相同。

!!! note “注意事项” 因为内存堆管理器要满足多线程情况下的安全分配，会考虑多线程间的互斥问题，所以请不要在中断服务例程中分配或释放动态内存块。因为它可能会引起当前上下文被挂起等待。

7.2.1 小内存管理算法

小内存管理算法是一个简单的内存分配算法。初始时，它是一块大的内存。当需要分配内存块时，将从这个大的内存块上分割出相匹配的内存块，然后把分割出来的空闲内存块还给堆管理系统中。每个内存块都包含一个管理用的数据头，通过这个头把使用块与空闲块用双向链表的方式链接起来，如下图所示：

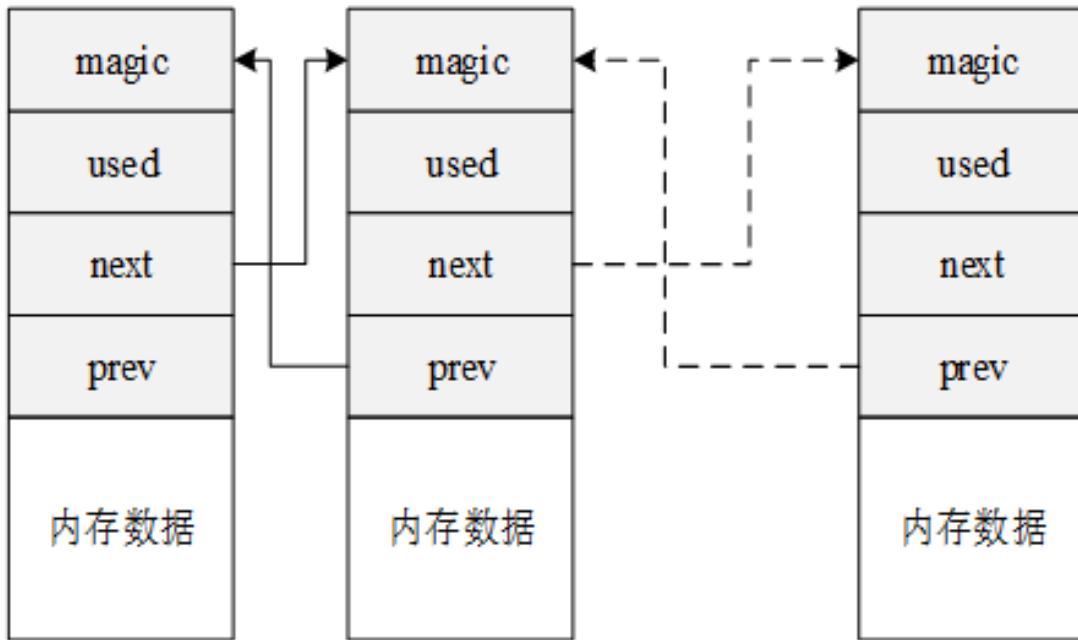


图 7.2: 小内存管理工作机制图

每个内存块（不管是已分配的内存块还是空闲的内存块）都包含一个数据头，其中包括：

1) **magic**: 变数（或称为幻数），它会被初始化成 0x1ea0（即英文单词 heap），用于标记这个内存块是一个内存管理用的内存数据块；变数不仅仅用于标识这个数据块是一个内存管理用的内存数据块，实质也是一个内存保护字：如果这个区域被改写，那么也就意味着这块内存块被非法改写（正常情况下只有内存管理器才会去碰这块内存）。

2) **used**: 指示出当前内存块是否已经分配。

内存管理的表现主要体现在内存的分配与释放上，小型内存管理算法可以用以下例子体现出来。

如下图所示的内存分配情况，空闲链表指针 **lfree** 初始指向 32 字节的内存块。当用户线程要再分配一个 64 字节的内存块时，但此 **lfree** 指针指向的内存块只有 32 字节并不能满足要求，内存管理器会继续寻找下一内存块，当找到再下一块内存块，128 字节时，它满足分配的要求。因为这个内存块比较大，分配器将把此内存块进行拆分，余下的内存块（52 字节）继续留在 **lfree** 链表中，如下图分配 64 字节后的链表结构所示。

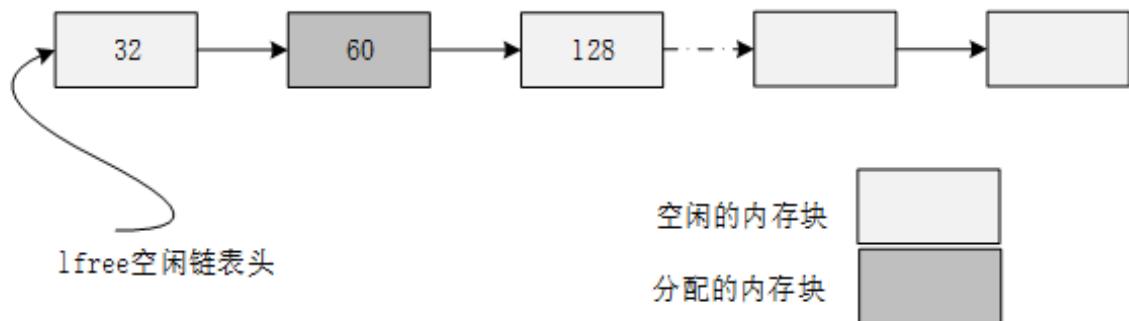


图 7.3: 小内存管理算法链表结构示意图 1

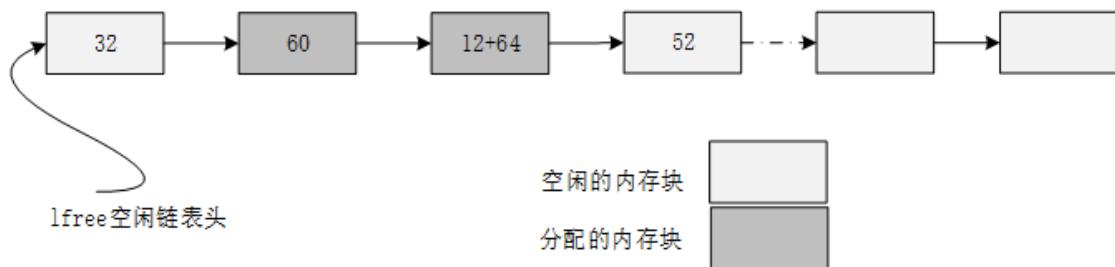


图 7.4: 小内存管理算法链表结构示意图 2

另外，在每次分配内存块前，都会留出 12 字节数据头用于 `magic`、`used` 信息及链表节点使用。返回给应用的地址实际上是这块内存块 12 字节以后的地址，前面的 12 字节数据头是用户永远不应该碰的部分（注：12 字节数据头长度会与系统对齐差异而有所不同）。

释放时则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

7.2.2 slab 管理算法

RT-Thread 的 slab 分配器是在 DragonFly BSD 创始人 Matthew Dillon 实现的 slab 分配器基础上，针对嵌入式系统优化的内存分配算法。最原始的 slab 算法是 Jeff Bonwick 为 Solaris 操作系统而引入的一种高效内核内存分配算法。

RT-Thread 的 slab 分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。slab 分配器会根据对象的大小分成多个区（zone），也可以看成每类对象有一个内存池，如下图所示：

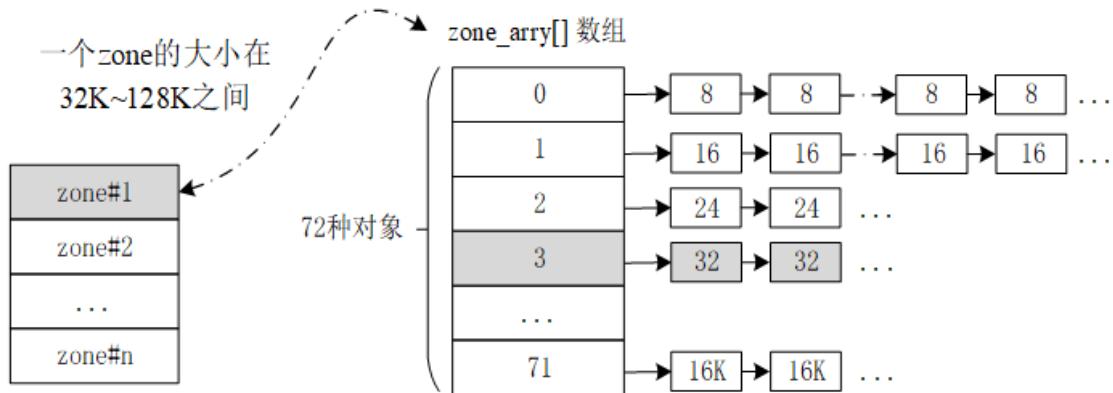


图 7.5: slab 内存分配结构图

一个 zone 的大小在 32K 到 128K 字节之间，分配器会在堆初始化时根据堆的大小自动调整。系统中的 zone 最多包括 72 种对象，一次最大能够分配 16K 的内存空间，如果超出了 16K 那么直接从页分配器中分配。每个 zone 上分配的内存块大小是固定的，能够分配相同大小内存块的 zone 会链接在一个链表中，而 72 种对象的 zone 链表则放在一个数组（zone_array[]）中统一管理。

下面是内存分配器主要的两种操作：

(1) 内存分配

假设分配一个 32 字节的内存，slab 内存分配器会先按照 32 字节的值，从 zone array 链表表头数组中找到相应的 zone 链表。如果这个链表是空的，则向页分配器分配一个新的 zone，然后从 zone 中返回第一个空闲内存块。如果链表非空，则这个 zone 链表中的第一个 zone 节点必然有空闲块存在（否则它就不应该放在这个链表中），那么就取相应的空闲块。如果分配完成后，zone 中所有空闲内存块都使用完毕，那么分配器需要把这个 zone 节点从链表中删除。

(2) 内存释放

分配器需要找到内存块所在的 zone 节点，然后把内存块链接到 zone 的空闲内存块链表中。如果此时 zone 的空闲链表指示出 zone 的所有内存块都已经释放，即 zone 是完全空闲的，那么当 zone 链表中全空闲 zone 达到一定数目后，系统就会把这个全空闲的 zone 释放到页面分配器中去。

7.2.3 memheap 管理算法

memheap 管理算法适用于系统含有多个地址可不连续的内存堆。使用 memheap 内存管理可以简化系统存在多个内存堆时的使用：当系统中存在多个内存堆的时候，用户只需要在系统初始化时将多个所需的 memheap 初始化，并开启 memheap 功能就可以很方便地把多个 memheap（地址可不连续）粘合起来用于系统的 heap 分配。

!!! note “注意事项” 在开启 memheap 之后原来的 heap 功能将被关闭，两者只可以通过打开或关闭 RT_USING_MEMHEAP_AS_HEAP 来选择其一

memheap 工作机制如下图所示，首先将多块内存加入 memheap_item 链表进行粘合。当分配内存块时，会先从默认内存堆去分配内存，当分配不到时会查找 memheap_item 链表，尝试从其他的内存堆上分配内存块。应用程序不用关心当前分配的内存块位于哪个内存堆上，就像是在操作一个内存堆。

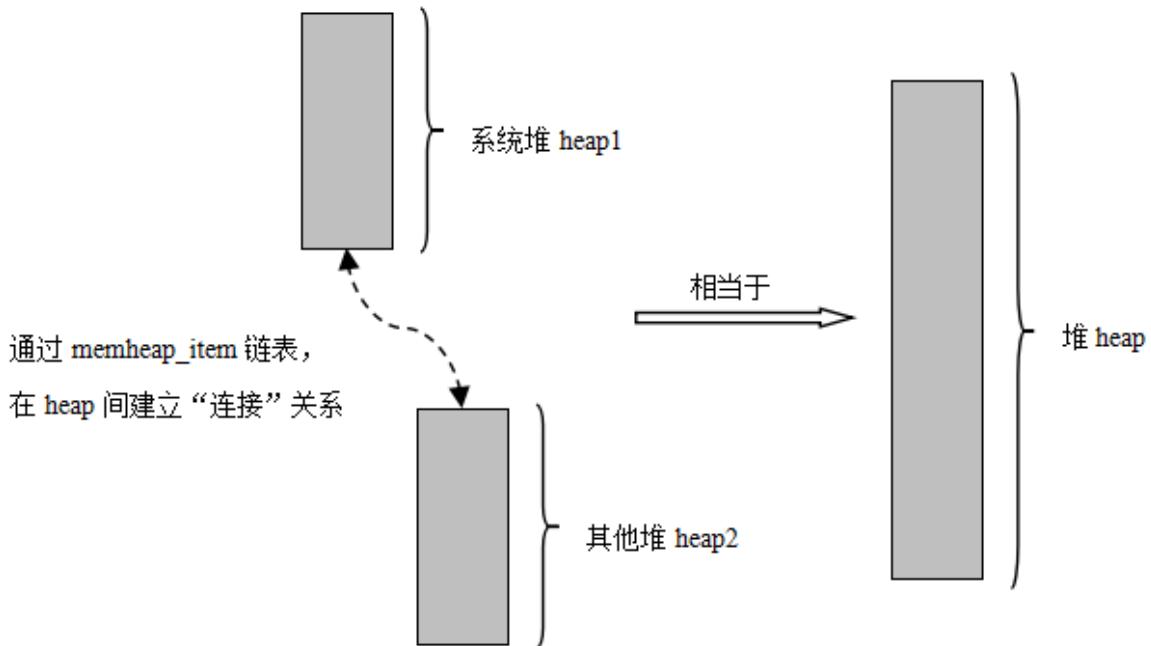


图 7.6: memheap 处理多内存堆

7.2.4 内存堆配置和初始化

在使用内存堆时，必须要在系统初始化的时候进行堆的初始化，可以通过下面的函数接口完成：

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

这个函数会把参数 `begin_addr`, `end_addr` 区域的内存空间作为内存堆来使用。下表描述了该函数的输入参数：

`rt_system_heap_init()` 的输入参数

参数	描述
<code>begin_addr</code>	堆内存区域起始地址
<code>end_addr</code>	堆内存区域结束地址

在使用 `memheap` 堆内存时，必须要在系统初始化的时候进行堆内存的初始化，可以通过下面的函数接口完成：

```
rt_err_t rt_memheap_init(struct rt_memheap *memheap,
                         const char *name,
                         void      *start_addr,
                         rt_uint32_t size)
```

如果有多个不连续的 `memheap` 可以多次调用该函数将其初始化并加入 `memheap_item` 链表。下表描述了该函数的输入参数与返回值：

`rt_memheap_init()` 的输入参数与返回值

参数	描述
memheap	memheap 控制块
name	内存堆的名称
start_addr	堆内存区域起始地址
size	堆内存大小
返回	—
RT_EOK	成功

7.2.5 内存堆的管理方式

对内存堆的操作如下图所示，包含：初始化、申请内存块、释放内存，所有使用完成后的动态内存都应该被释放，以供其他程序的申请使用。

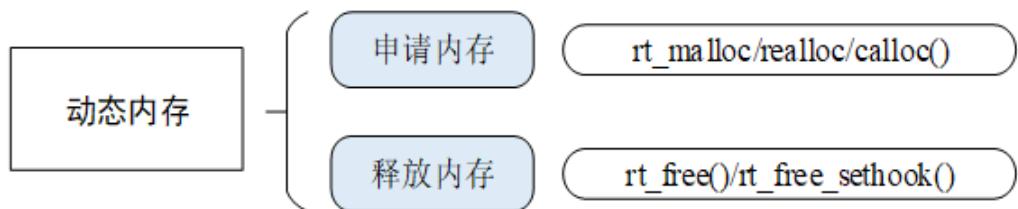


图 7.7: 内存堆的操作

7.2.5.1 分配和释放内存块

从内存堆上分配用户指定大小的内存块，函数接口如下：

```
void *rt_malloc(rt_size_t nbytes);
```

`rt_malloc` 函数会从系统堆空间中找到合适大小的内存块，然后把内存块可用地址返回给用户。下表描述了该函数的输入参数与返回值：

`rt_malloc()` 的输入参数和返回值

参数	描述
nbytes	需要分配的内存块的大小，单位为字节
返回	—
分配的内存块地址	成功
RT_NULL	失败

应用程序使用完从内存分配器中申请的内存后，必须及时释放，否则会造成内存泄漏，释放内存块的函数接口如下：

```
void rt_free (void *ptr);
```

`rt_free` 函数会把待释放的内存还给堆管理器中。在调用这个函数时用户需传递待释放的内存块指针，如果是空指针直接返回。下表描述了该函数的输入参数：

`rt_free()` 的输入参数

参数	描述
ptr	待释放的内存块指针

7.2.5.2 重分配内存块

在已分配内存块的基础上重新分配内存块的大小（增加或缩小），可以通过下面的函数接口完成：

```
void *rt_realloc(void *rmem, rt_size_t newsize);
```

在进行重新分配内存块时，原来的内存块数据保持不变（缩小的情况下，后面的数据被自动截断）。下表描述了该函数的输入参数和返回值：

`rt_realloc()` 的输入参数和返回值

参数	描述
rmem	指向已分配的内存块
newsize	重新分配的内存大小
返回	—
重新分配的内存块地址	成功

7.2.5.3 分配多内存块

从内存堆中分配连续内存地址的多个内存块，可以通过下面的函数接口完成：

```
void *rt_calloc(rt_size_t count, rt_size_t size);
```

下表描述了该函数的输入参数与返回值：

`rt_calloc()` 的输入参数和返回值

参数	描述
count	内存块数量
size	内存块容量
返回	—

参数	描述
指向第一个内存块地址的指针	成功，并且所有分配的内存块都被初始化成零。
RT_NULL	分配失败

7.2.5.4 设置内存钩子函数

在分配内存块过程中，用户可设置一个钩子函数，调用的函数接口如下：

```
void rt_malloc_sethook(void (*hook)(void *ptr, rt_size_t size));
```

设置的钩子函数会在内存分配完成后进行回调。回调时，会把分配到的内存块地址和大小做为入口参数传递进去。下表描述了该函数的输入参数：

rt_malloc_sethook() 的输入参数

参数	描述
hook	钩子函数指针

其中 hook 函数接口如下：

```
void hook(void *ptr, rt_size_t size);
```

下表描述了 hook 函数的输入参数：

分配钩子 hook 函数接口参数

参数	描述
ptr	分配到的内存块指针
size	分配到的内存块的大小

在释放内存时，用户可设置一个钩子函数，调用的函数接口如下：

```
void rt_free_sethook(void (*hook)(void *ptr));
```

设置的钩子函数会在调用内存释放完成前进行回调。回调时，释放的内存块地址会做为入口参数传递进去（此时内存块并没有被释放）。下表描述了该函数的输入参数：

rt_free_sethook() 的输入参数

参数	描述
hook	钩子函数指针

其中 hook 函数接口如下：

```
void hook(void *ptr);
```

下表描述了 hook 函数的输入参数：

钩子函数 hook 的输入参数

参数	描述
ptr	待释放的内存块指针

7.2.6 内存堆管理应用示例

这是一个内存堆的应用示例，这个程序会创建一个动态的线程，这个线程会动态申请内存并释放，每次申请更大的内存，当申请不到的时候就结束，如下代码所示：

内存堆管理

```
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

/* 线程入口 */
void thread1_entry(void *parameter)
{
    int i;
    char *ptr = RT_NULL; /* 内存块的指针 */

    for (i = 0; ; i++)
    {
        /* 每次分配 (1 << i) 大小字节数的内存空间 */
        ptr = rt_malloc(1 << i);

        /* 如果分配成功 */
        if (ptr != RT_NULL)
        {
            rt_kprintf("get memory :%d byte\n", (1 << i));
            /* 释放内存块 */
            rt_free(ptr);
            rt_kprintf("free memory :%d byte\n", (1 << i));
            ptr = RT_NULL;
        }
        else
        {
            rt_kprintf("try to get %d byte memory failed!\n", (1 << i));
            return;
        }
    }
}
```

```

        }
    }

int dynmem_sample(void)
{
    rt_thread_t tid = RT_NULL;

    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                          thread1_entry, RT_NULL,
                          THREAD_STACK_SIZE,
                          THREAD_PRIORITY,
                          THREAD_TIMESLICE);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(dynmem_sample, dynmem sample);

```

仿真运行结果如下：

```

\ | /
- RT -      Thread Operating System
 / | \  3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >dynmem_sample
msh >get memory :1 byte
free memory :1 byte
get memory :2 byte
free memory :2 byte
...
get memory :16384 byte
free memory :16384 byte
get memory :32768 byte
free memory :32768 byte
try to get 65536 byte memory failed!

```

例程中分配内存成功并打印信息；当试图申请 65536 byte 即 64KB 内存时，由于 RAM 总大小只有 64K，而可用 RAM 小于 64K，所以分配失败。

7.3 内存池

内存堆管理器可以分配任意大小的内存块，非常灵活和方便。但其也存在明显的缺点：一是分配效率不高，在每次分配时，都要空闲内存块查找；二是容易产生内存碎片。为了提高内存分配的效率，并且避免内存碎片，RT-Thread 提供了另外一种内存管理方法：内存池（Memory Pool）。

内存池是一种内存分配方式，用于分配大量大小相同的小内存块，它可以极大地加快内存分配与释放的速度，且能尽量避免内存碎片化。此外，RT-Thread 的内存池支持线程挂起功能，当内存池中无空闲内存块时，申请线程会被挂起，直到内存池中有新的可用内存块，再将挂起的申请线程唤醒。

内存池的线程挂起功能非常适合需要通过内存资源进行同步的场景，例如播放音乐时，播放器线程会对音乐文件进行解码，然后发送到声卡驱动，从而驱动硬件播放音乐。

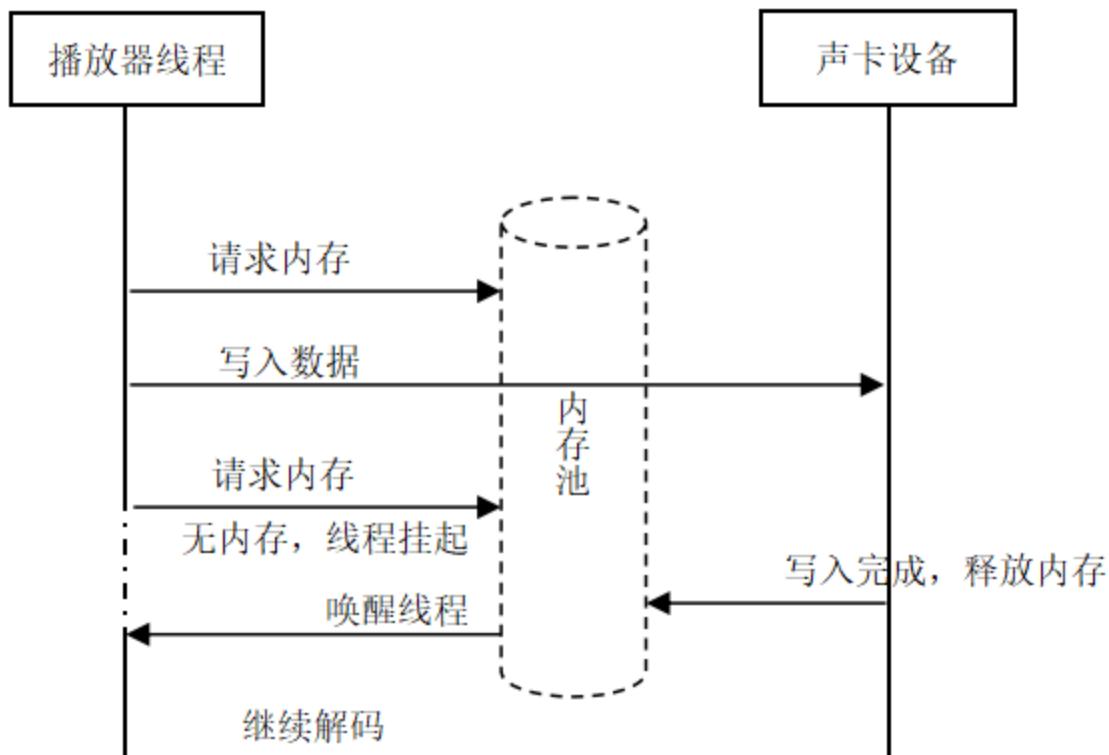


图 7.8: 播放器线程与声卡驱动关系

如上图所示，当播放器线程需要解码数据时，就会向内存池请求内存块，如果内存块已经用完，线程将被挂起，否则它将获得内存块以放置解码的数据；

而后播放器线程把包含解码数据的内存块写入到声卡抽象设备中（线程会立刻返回，继续解码出更多的数据）；

当声卡设备写入完成后，将调用播放器线程设置的回调函数，释放写入的内存块，如果在此之前，播放器线程因为把内存池里的内存块都用完而被挂起的话，那么这是它将被唤醒，并继续进行解码。

7.3.1 内存池工作机制

7.3.1.1 内存池控制块

内存池控制块是操作系统用于管理内存池的一个数据结构，它会存放内存池的一些信息，例如内存池数据区域开始地址，内存块大小和内存块列表等，也包含内存块与内存块之间连接用的链表结构，因内存块不可用而挂起的线程等待事件集合等。

在 RT-Thread 实时操作系统中，内存池控制块由结构体 struct rt_mempool 表示。另外一种 C 表达方式 rt_mp_t，表示的是内存块句柄，在 C 语言中的实现是指向内存池控制块的指针，详细定义情况见以下代码：

```
struct rt_mempool
{
    struct rt_object parent;

    void *start_address; /* 内存池数据区域开始地址 */
    rt_size_t size; /* 内存池数据区域大小 */

    rt_size_t block_size; /* 内存块大小 */
    rt_uint8_t *block_list; /* 内存块列表 */

    /* 内存池数据区域中能够容纳的最大内存块数 */
    rt_size_t block_total_count;
    /* 内存池中空闲的内存块数 */
    rt_size_t block_free_count;
    /* 因为内存块不可用而挂起的线程列表 */
    rt_list_t suspend_thread;
    /* 因为内存块不可用而挂起的线程数 */
    rt_size_t suspend_thread_count;
};

typedef struct rt_mempool* rt_mp_t;
```

7.3.1.2 内存块分配机制

内存池在创建时先向系统申请一大块内存，然后分成同样大小的多个小内存块，小内存块直接通过链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出链头上第一个内存块，提供给申请者。从下图中可以看到，物理内存中允许存在多个大小不同的内存池，每一个内存池又由多个空闲内存块组成，内核用它们来进行内存管理。当一个内存池对象被创建时，内存池对象就被分配给了一个内存池控制块，内存控制块的参数包括内存池名，内存缓冲区，内存块大小，块数以及一个等待线程队列。

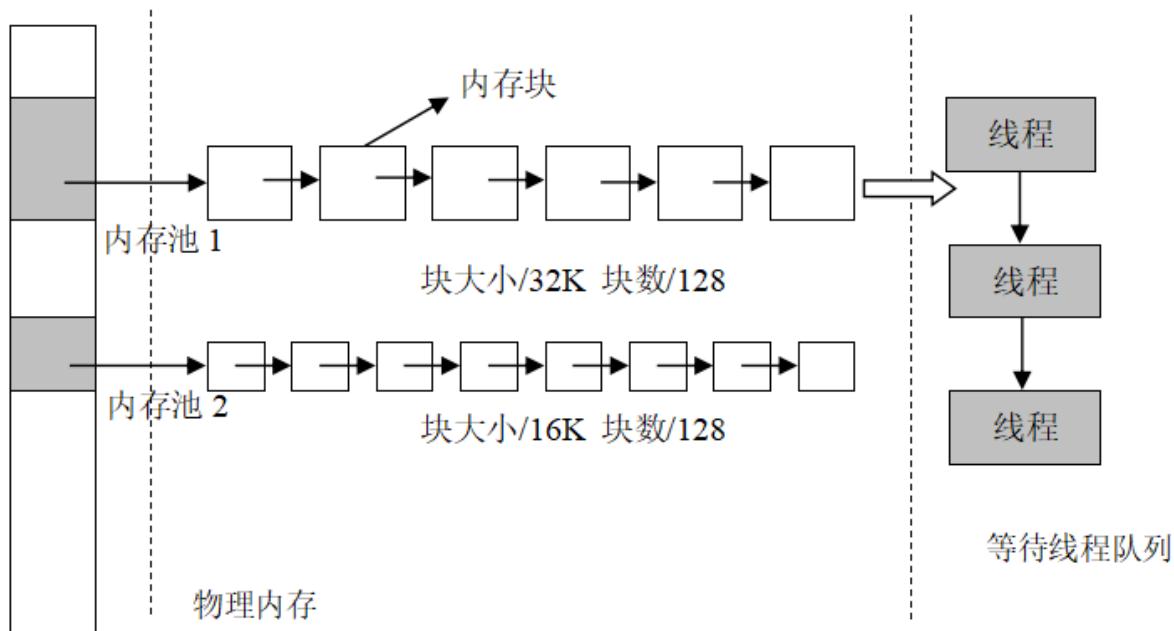


图 7.9: 内存池工作机制图

内核负责给内存池分配内存池控制块，它同时也接收用户线程的分配内存块申请，当获得这些信息后，内核就可以从内存池中为内存池分配内存。内存池一旦初始化完成，内部的内存块大小将不能再做调整。

每一个内存池对象由上述结构组成，其中 `suspend_thread` 形成了一个申请线程等待列表，即当内存池中无可用内存块，并且申请线程允许等待时，申请线程将挂起在 `suspend_thread` 链表上。

7.3.2 内存池的管理方式

内存池控制块是一个结构体，其中含有内存池相关的重要参数，在内存池各种状态间起到纽带的作用。内存池的相关接口如下图所示，对内存池的操作包含：创建 / 初始化内存池、申请内存块、释放内存块、删除 / 脱离内存池，但不是所有的内存池都会被删除，这与设计者的需求相关，但是使用完的内存块都应该被释放。

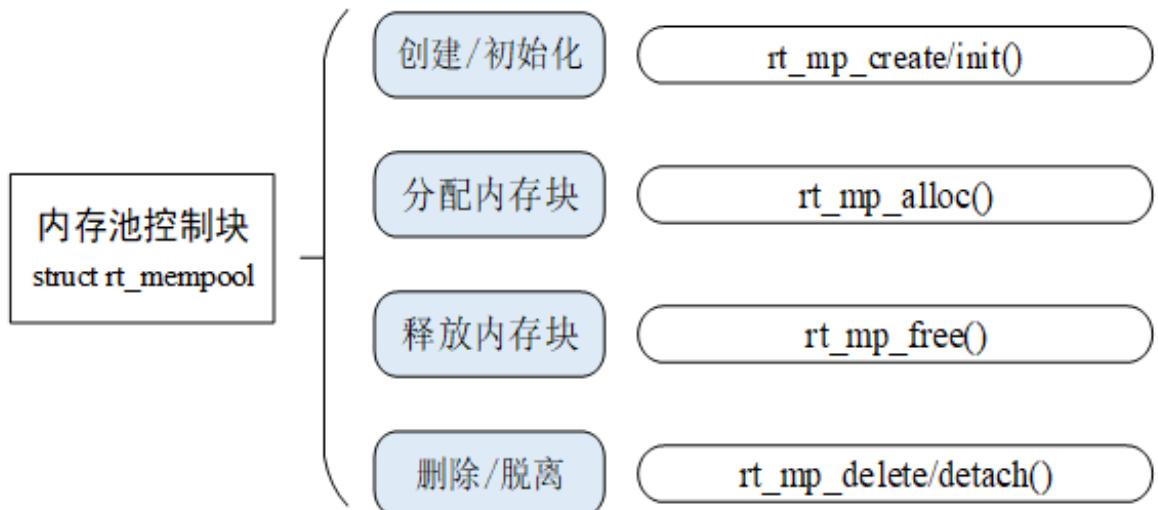


图 7.10: 内存池相关接口

7.3.2.1 创建和删除内存池

创建内存池操作将会创建一个内存池对象并从堆上分配一个内存池。创建内存池是从对应内存池中分配和释放内存块的先决条件，创建内存池后，线程便可以从内存池中执行申请、释放等操作。创建内存池使用下面的函数接口，该函数返回一个已创建的内存池对象。

```
rt_mp_t rt_mp_create(const char* name,
                      rt_size_t block_count,
                      rt_size_t block_size);
```

使用该函数接口可以创建一个与需求的内存块大小、数目相匹配的内存池，前提当然是在系统资源允许的情况下（最主要的是内存堆内存资源）才能创建成功。创建内存池时，需要给内存池指定一个名称。然后内核从系统中申请一个内存池对象，然后从内存堆中分配一块由块数目和块大小计算得来的内存缓冲区，接着初始化内存池对象，并将申请成功的内存缓冲区组织成可用于分配的空闲块链表。下表描述了该函数的输入参数与返回值：

`rt_mp_create()` 的输入参数和返回值

参数	描述
<code>name</code>	内存池名
<code>block_count</code>	内存块数量
<code>block_size</code>	内存块容量
返回	—
内存池的句柄	创建内存池对象成功
<code>RT_NULL</code>	创建失败

删除内存池将删除内存池对象并释放申请的内存。使用下面的函数接口：

```
rt_err_t rt_mp_delete(rt_mp_t mp);
```

删除内存池时，会首先唤醒等待在该内存池对象上的所有线程（返回 -RT_ERROR），然后再释放已从内存堆上分配的内存池数据存放区域，然后删除内存池对象。下表描述了该函数的输入参数与返回值：

`rt_mp_delete()` 的输入参数和返回值

参数	描述
mp	<code>rt_mp_create</code> 返回的内存池对象句柄
返回	—
RT_EOK	删除成功

7.3.2.2 初始化和脱离内存池

初始化内存池跟创建内存池类似，只是初始化内存池用于静态内存管理模式，内存池控制块来源于用户在系统中申请的静态对象。另外与创建内存池不同的是，此处内存池对象所使用的内存空间是由用户指定的一个缓冲区空间，用户把缓冲区的指针传递给内存池控制块，其余的初始化工作与创建内存池相同。函数接口如下：

```
rt_err_t rt_mp_init(rt_mp_t mp,
                    const char* name,
                    void *start, rt_size_t size,
                    rt_size_t block_size);
```

初始化内存池时，把需要进行初始化的内存池对象传递给内核，同时需要传递的还有内存池用到的内存空间，以及内存池管理的内存块数目和块大小，并且给内存池指定一个名称。这样，内核就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块链表。下表描述了该函数的输入参数与返回值：

`rt_mp_init()` 的输入参数和返回值

参数	描述
mp	内存池对象
name	内存池名
start	内存池的起始位置
size	内存池数据区域大小
block_size	内存块容量
返回	—
RT_EOK	初始化成功
- RT_ERROR	失败

内存池块个数 = $\text{size} / (\text{block_size} + 4 \text{ 链表指针大小})$, 计算结果取整数。

例如：内存池数据区总大小 `size` 设为 4096 字节，内存块大小 `block_size` 设为 80 字节；则申请的内存块个数为 $4096 / (80+4) = 48$ 个。

脱离内存池将把内存池对象从内核对象管理器中脱离。脱离内存池使用下面的函数接口：

```
rt_err_t rt_mp_detach(rt_mp_t mp);
```

使用该函数接口后，内核先唤醒所有等待在该内存池对象上的线程，然后将内存池对象从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

`rt_mp_detach()` 的输入参数和返回值

参数	描述
<code>mp</code>	内存池对象
返回	—
<code>RT_EOK</code>	成功

7.3.2.3 分配和释放内存块

从指定的内存池中分配一个内存块，使用如下接口：

```
void *rt_mp_alloc (rt_mp_t mp, rt_int32_t time);
```

其中 `time` 参数的含义是申请分配内存块的超时时间。如果内存池中有可用的内存块，则从内存池的空闲块链表上取下一个内存块，减少空闲块数目并返回这个内存块；如果内存池中已经没有空闲内存块，则判断超时时间设置：若超时时间设置为零，则立刻返回空内存块；若等待时间大于零，则把当前线程挂起在该内存池对象上，直到内存池中有可用的自由内存块，或等待时间到达。下表描述了该函数的输入参数与返回值：

`rt_mp_alloc()` 的输入参数和返回值

参数	描述
<code>mp</code>	内存池对象
<code>time</code>	超时时间
返回	—
分配的内存块地址	成功
<code>RT_NULL</code>	失败

任何内存块使用完后都必须被释放，否则会造成内存泄露，释放内存块使用如下接口：

```
void rt_mp_free (void *block);
```

使用该函数接口时，首先通过需要被释放的内存块指针计算出该内存块所在的（或所属于的）内存池

对象，然后增加内存池对象的可用内存块数目，并把该被释放的内存块加入空闲内存块链表上。接着判断该内存池对象上是否有挂起的线程，如果有，则唤醒挂起线程链表上的首线程。下表描述了该函数的输入参数：

rt_mp_free() 的输入参数

参数	描述
block	内存块指针

7.3.3 内存池应用示例

这是一个静态内内存池的应用例程，这个例程会创建一个静态的内存池对象，2个动态线程。一个线程会试图从内存池中获得内存块，另一个线程释放内存块内存块，如下代码所示：

内存池使用示例

```
#include <rtthread.h>

static rt_uint8_t *ptr[50];
static rt_uint8_t mempool[4096];
static struct rt_mempool mp;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

/* 线程 1 入口 */
static void thread1_mp_alloc(void *parameter)
{
    int i;
    for (i = 0 ; i < 50 ; i++)
    {
        if (ptr[i] == RT_NULL)
        {
            /* 试图申请内存块 50 次，当申请不到内存块时，
               线程 1 挂起，转至线程 2 运行 */
            ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
            if (ptr[i] != RT_NULL)
                rt_kprintf("allocate No.%d\n", i);
        }
    }
}

/* 线程 2 入口，线程 2 的优先级比线程 1 低，应该线程 1 先获得执行。*/
static void thread2_mp_free(void *parameter)
{
    int i;
    for (i = 0 ; i < 50 ; i++)
    {
        if (ptr[i] != RT_NULL)
            rt_mp_free(ptr[i], &mp);
    }
}
```

```
static void thread2_mp_release(void *parameter)
{
    int i;

    rt_kprintf("thread2 try to release block\n");
    for (i = 0; i < 50 ; i++)
    {
        /* 释放所有分配成功的内存块 */
        if (ptr[i] != RT_NULL)
        {
            rt_kprintf("release block %d\n", i);
            rt_mp_free(ptr[i]);
            ptr[i] = RT_NULL;
        }
    }
}

int mempool_sample(void)
{
    int i;
    for (i = 0; i < 50; i++) ptr[i] = RT_NULL;

    /* 初始化内存池对象 */
    rt_mp_init(&mp, "mp1", &mempool[0], sizeof(mempool), 80);

    /* 创建线程 1：申请内存池 */
    tid1 = rt_thread_create("thread1", thread1_mp_alloc, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 创建线程 2：释放内存池 */
    tid2 = rt_thread_create("thread2", thread2_mp_release, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(mempool_sample, mempool sample);
```

仿真运行结果如下：

\ | /

```
- RT -      Thread Operating System
/ | \    3.1.0 build Aug 24 2018
2006 - 2018 Copyright by rt-thread team
msh >mempool_sample
msh >allocate No.0
allocate No.1
allocate No.2
allocate No.3
allocate No.4
...
allocate No.46
allocate No.47
thread2 try to release block
release block 0
allocate No.48
release block 1
allocate No.49
release block 2
release block 3
release block 4
release block 5
...
release block 47
release block 48
release block 49
```

本例程在初始化内存池对象时，初始化了 $4096 / (80+4) = 48$ 个内存块。

- 线程 1 申请了 48 个内存块之后，此时内存块已经被用完，需要其他地方释放才能再次申请；但此时，线程 1 以一直等待的方式又申请了 1 个，由于无法分配，所以线程 1 挂起；
- 线程 2 开始执行释放内存的操作；当线程 2 释放一个内存块的时候，就有一个内存块空闲出来，唤醒线程 1 申请内存，申请成功后再申请，线程 1 又挂起，再循环一次 □；
- 线程 2 继续释放剩余的内存块，释放完毕。

第8章

中断管理

什么是中断？简单的解释就是系统正在处理某一个正常事件，忽然被另一个需要马上处理的紧急事件打断，系统转而处理这个紧急事件，待处理完毕，再恢复运行刚才被打断的事件。生活中，我们经常会遇到这样的场景：

当你正在专心看书的时候，忽然来了一个电话，于是记下书的页码，去接电话，接完电话后接着刚才的页码继续看书，这是一个典型的中断的过程。

电话是老师打过来的，让你赶快交作业，你判断交作业的优先级比看书高，于是电话挂断后先做作业，等交完作业后再接着刚才的页码继续看书，这是一个典型的在中断中进行任务调度的过程。

这些场景在嵌入式系统中也很常见，当 CPU 正在处理内部数据时，外界发生了紧急情况，要求 CPU 暂停当前的工作转去处理这个 [异步事件](#)。处理完毕后，再回到原来被中断的地址，继续原来的工作，这样的过程称为中断。实现这一功能的系统称为 [中断系统](#)，申请 CPU 中断的请求源称为 [中断源](#)。中断是一种异常，异常是导致处理器脱离正常运行转向执行特殊代码的任何事件，如果不及时进行处理，轻则系统出错，重则会导致系统毁灭性地瘫痪。所以正确地处理异常，避免错误的发生是提高软件鲁棒性（稳定性）非常重要的一环。如下图是一个简单的中断示意图。

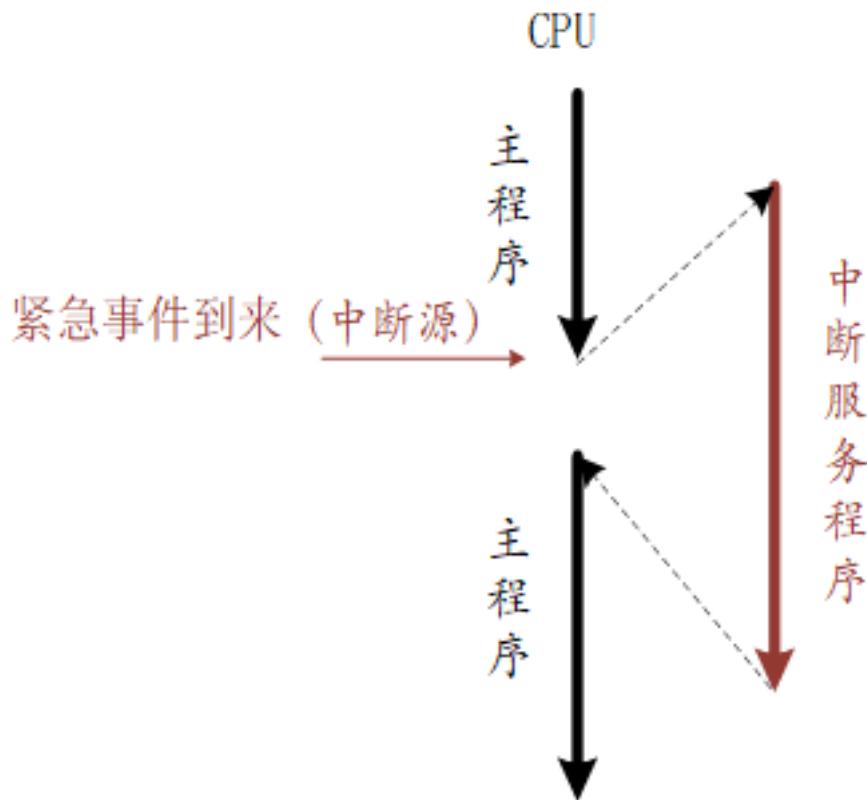


图 8.1: 中断示意图

中断处理与 CPU 架构密切相关，所以，本章会先介绍 ARM Cortex-M 的 CPU 架构，然后结合 Cortex-M CPU 架构来介绍 RT-Thread 的中断管理机制，读完本章，大家将深入了解 RT-Thread 的中断处理过程，如何添加中断服务程序（ISR）以及相关的注意事项。

8.1 Cortex-M CPU 架构基础

不同于老的经典 ARM 处理器（例如：ARM7, ARM9），ARM Cortex-M 处理器有一个非常不同的架构，Cortex-M 是一个家族系列，其中包括 Cortex M0/M3/M4/M7 多个不同型号，每个型号之间会有些区别，例如 Cortex-M4 比 Cortex-M3 多了浮点计算功能等，但它们的编程模型基本是一致的，因此本书中介绍中断管理和移植的部分都不会对 Cortex M0/M3/M4/M7 做太精细的区分。本节主要介绍和 RT-Thread 中断管理相关的架构部分。

8.1.1 寄存器简介

Cortex-M 系列 CPU 的寄存器组里有 R0~R15 共 16 个通用寄存器组和若干特殊功能寄存器，如下图所示。

通用寄存器组里的 R13 作为堆栈指针寄存器 (Stack Pointer, SP); R14 作为连接寄存器 (Link Register, LR)，用于在调用子程序时，存储返回地址；R15 作为程序计数器 (Program Counter, PC)，其中堆栈指针寄存器可以是主堆栈指针 (MSP)，也可以是进程堆栈指针 (PSP)。

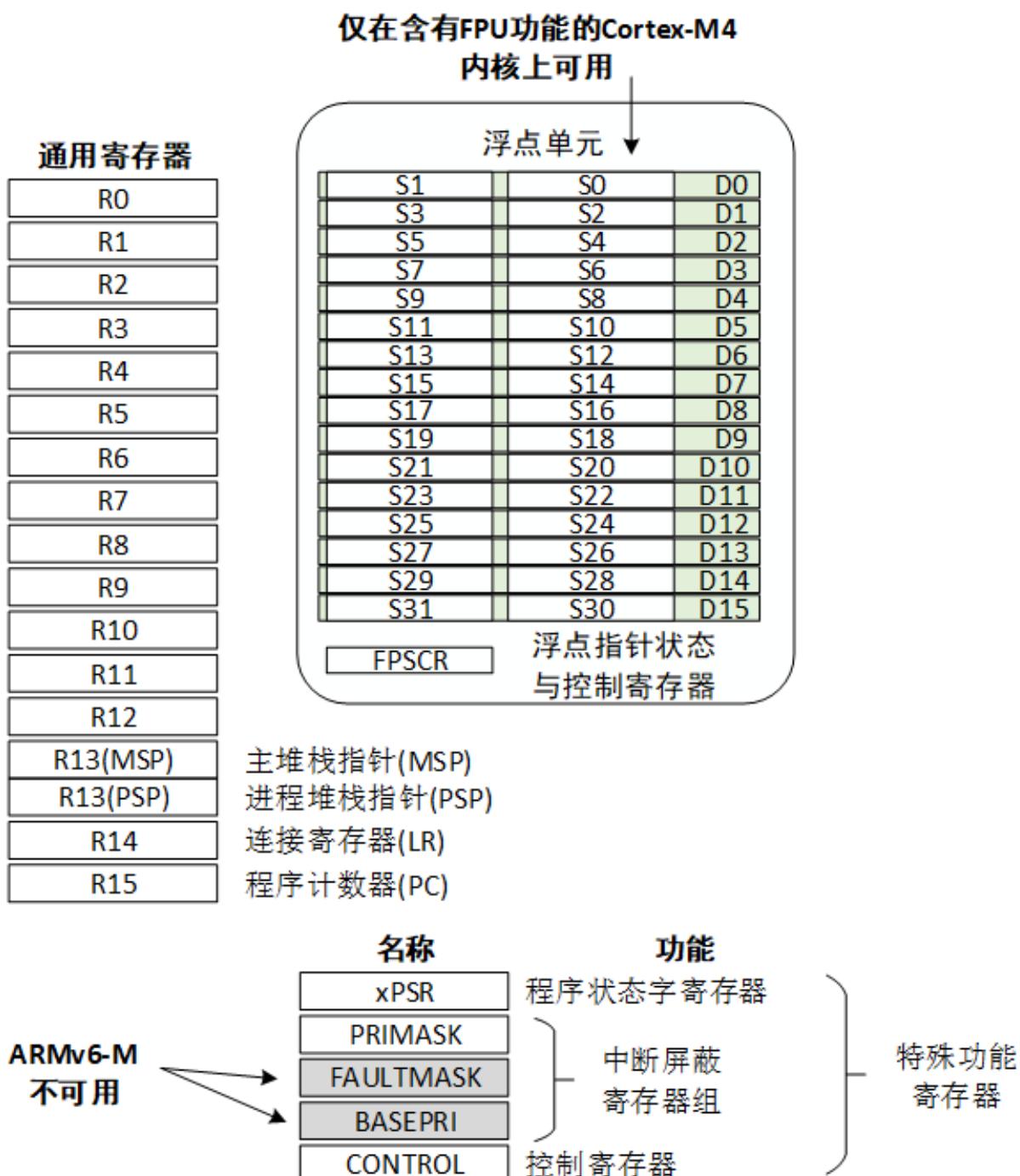


图 8.2: Cortex-M 寄存器示意图

特殊功能寄存器包括程序状态字寄存器组 (PSRs)、中断屏蔽寄存器组 (PRIMASK, FAULTMASK, BASEPRI)、控制寄存器 (CONTROL)，可以通过 MSR/MRS 指令来访问特殊功能寄存器，例如：

```
MRS R0, CONTROL ; 读取 CONTROL 到 R0 中
MSR CONTROL, R0 ; 写入 R0 到 CONTROL 寄存器中
```

程序状态字寄存器里保存算术与逻辑标志，例如负数标志，零结果标志，溢出标志等等。中断屏蔽寄存器组控制 Cortex-M 的中断除能。控制寄存器用来定义特权级别和当前使用哪个堆栈指针。

如果是具有浮点单元的 Cortex-M4 或者 Cortex-M7，控制寄存器也用来指示浮点单元当前是否在使用，浮点单元包含了 32 个浮点通用寄存器 S0~S31 和特殊 FPSCR 寄存器（Floating point status and control register）。

8.1.2 操作模式和特权级别

Cortex-M 引入了操作模式和特权级别的概念，分别为线程模式和处理模式，如果进入异常或中断处理则进入处理模式，其他情况则为线程模式。

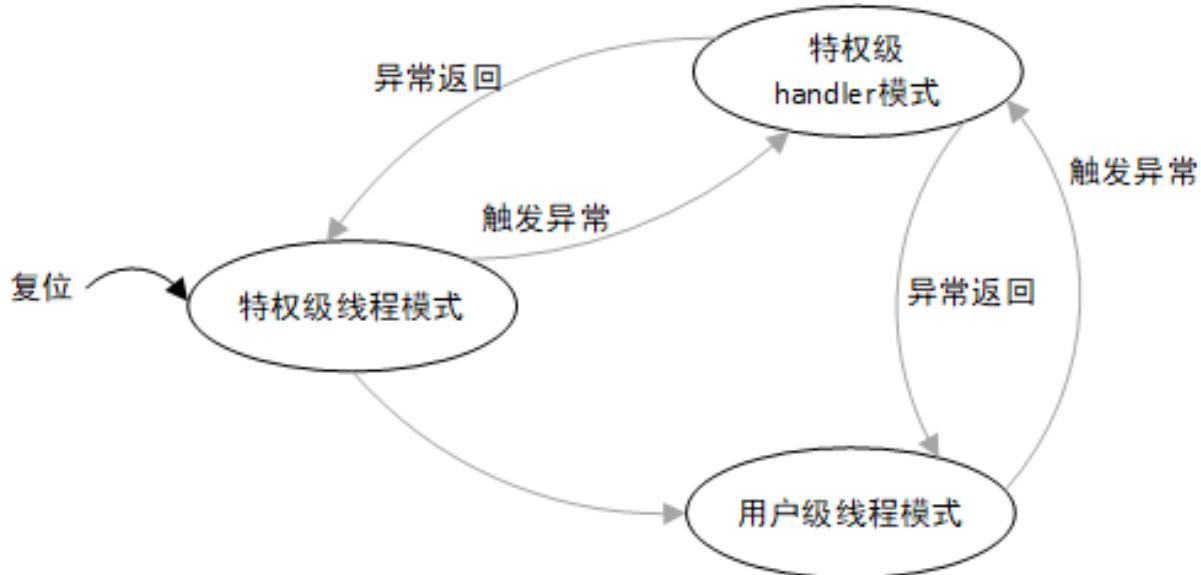


图 8.3: Cortex-M 工作模式状态图

Cortex-M 有两个运行级别，分别为特权级和用户级，线程模式可以工作在特权级或者用户级，而处理模式总工作在特权级，可通过 CONTROL 特殊寄存器控制。工作模式状态切换情况如上图所示。

Cortex-M 的堆栈寄存器 SP 对应两个物理寄存器 MSP 和 PSP，MSP 为主堆栈，PSP 为进程堆栈，处理模式总是使用 MSP 作为堆栈，线程模式可以选择使用 MSP 或 PSP 作为堆栈，同样通过 CONTROL 特殊寄存器控制。复位后，Cortex-M 默认进入线程模式、特权级、使用 MSP 堆栈。

8.1.3 嵌套向量中断控制器

Cortex-M 中断控制器名为 NVIC（嵌套向量中断控制器），支持中断嵌套功能。当一个中断触发并且系统进行响应时，处理器硬件会将当前运行位置的上下文寄存器自动压入中断栈中，这部分的寄存器包括 PSR、PC、LR、R12、R3-R0 寄存器。

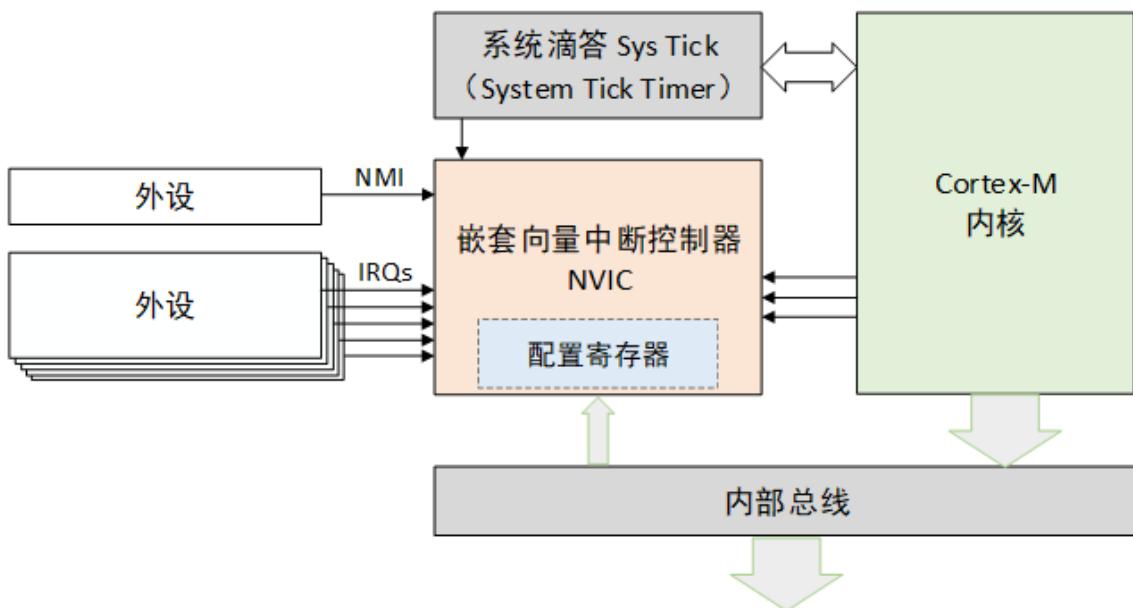


图 8.4: Cortex-M 内核和 NVIC 关系示意图

当系统正在服务一个中断时，如果有一个更高优先级的中断触发，那么处理器同样会打断当前运行的中断服务程序，然后把这个中断服务程序上下文的 PSR、PC、LR、R12、R3-R0 寄存器自动保存到中断栈中。

8.1.4 PendSV 系统调用

PendSV 也称为可悬起的系统调用，它是一种异常，可以像普通的中断一样被挂起，它是专门用来辅助操作系统进行上下文切换的。PendSV 异常会被初始化为最低优先级的异常。每次需要进行上下文切换的时候，会手动触发 PendSV 异常，在 PendSV 异常处理函数中进行上下文切换。在下一章《内核移植》中会详细介绍利用 PendSV 机制进行操作系统上下文切换的详细流程。

8.2 RT-Thread 中断工作机制

8.2.1 中断向量表

中断向量表是所有中断处理程序的入口，如下图所示是 Cortex-M 系列的中断处理过程：把一个函数（用户中断服务程序）同一个虚拟中断向量表中的中断向量联系在一起。当中断向量对应中断发生的时候，被挂接的用户中断服务程序就会被调用执行。

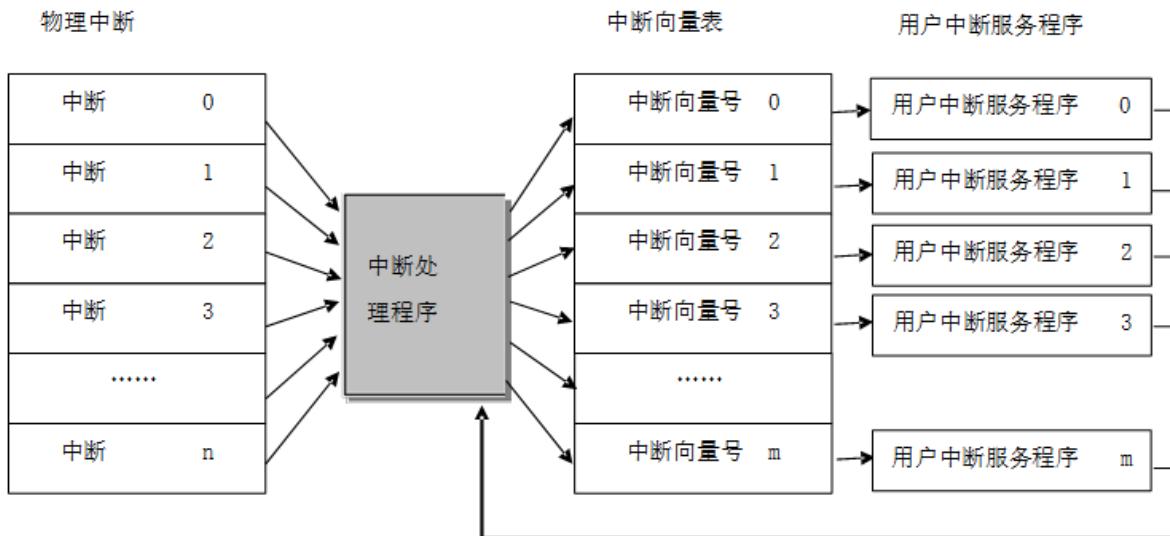


图 8.5: 中断处理过程

在 Cortex-M 内核上，所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理，每个中断服务程序必须排列在一起放在统一的地址上（这个地址必须要设置到 NVIC 的中断向量偏移寄存器中）。中断向量表一般由一个数组定义或在起始代码中给出，默认采用起始代码给出：

```

__Vectors    DCD      __initial_sp          ; Top of Stack
             DCD      Reset_Handler        ; Reset 处理函数
             DCD      NMI_Handler         ; NMI 处理函数
             DCD      HardFault_Handler   ; Hard Fault 处理函数
             DCD      MemManage_Handler   ; MPU Fault 处理函数
             DCD      BusFault_Handler   ; Bus Fault 处理函数
             DCD      UsageFault_Handler ; Usage Fault 处理函数
             DCD      0                  ; 保留
             DCD      0                  ; 保留
             DCD      0                  ; 保留
             DCD      0                  ; 保留
             DCD      SVC_Handler        ; SVCall 处理函数
             DCD      DebugMon_Handler   ; Debug Monitor 处理函数
             DCD      0                  ; 保留
             DCD      PendSV_Handler     ; PendSV 处理函数
             DCD      SysTick_Handler    ; SysTick 处理函数

...
NMI_Handler      PROC
                  EXPORT NMI_Handler       [WEAK]
                  B .
                  ENDP
HardFault_Handler  PROC
                  EXPORT HardFault_Handler [WEAK]

```

```

B
ENDP
...

```

请注意代码后面的 [WEAK] 标识，它是符号弱化标识，在 [WEAK] 前面的符号（如 NMI_Handler、HardFault_Handler）将被执行弱化处理，如果整个代码在链接时遇到了名称相同的符号（例如与 NMI_Handler 相同名称的函数），那么代码将使用未被弱化定义的符号（与 NMI_Handler 相同名称的函数），而与弱化符号相关的代码将被自动丢弃。

以 SysTick 中断为例，在系统启动代码中，需要填上 SysTick_Handler 中断入口函数，然后实现该函数即可对 SysTick 中断进行响应，中断处理函数示例程序如下所示：

```

void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}

```

8.2.2 中断处理过程

RT-Thread 中断管理中，将中断处理程序分为中断前导程序、用户中断服务程序、中断后续程序三部分，如下图：

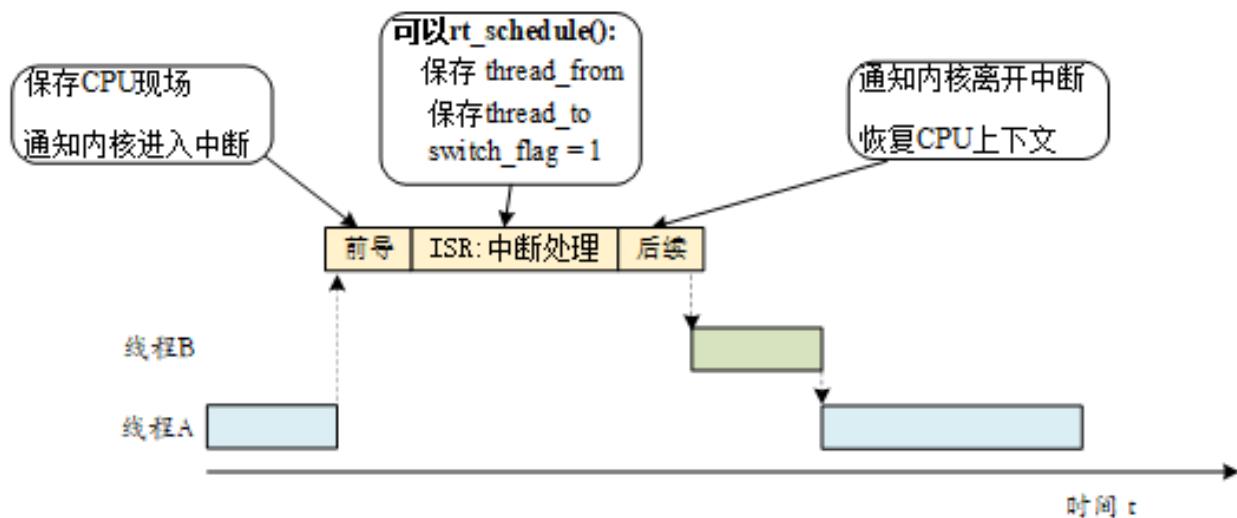


图 8.6：中断处理程序的 3 部分

8.2.2.1 中断前导程序

中断前导程序主要工作如下：

1) 保存 CPU 中断现场，这部分跟 CPU 架构相关，不同 CPU 架构的实现方式有差异。

对于 Cortex-M 来说，该工作由硬件自动完成。当一个中断触发并且系统进行响应时，处理器硬件会将当前运行部分的上下文寄存器自动压入中断栈中，这部分的寄存器包括 PSR、PC、LR、R12、R3-R0 寄存器。

2) 通知内核进入中断状态，调用 `rt_interrupt_enter()` 函数，作用是把全局变量 `rt_interrupt_nest` 加 1，用它来记录中断嵌套的层数，代码如下所示。

```
void rt_interrupt_enter(void)
{
    rt_base_t level;

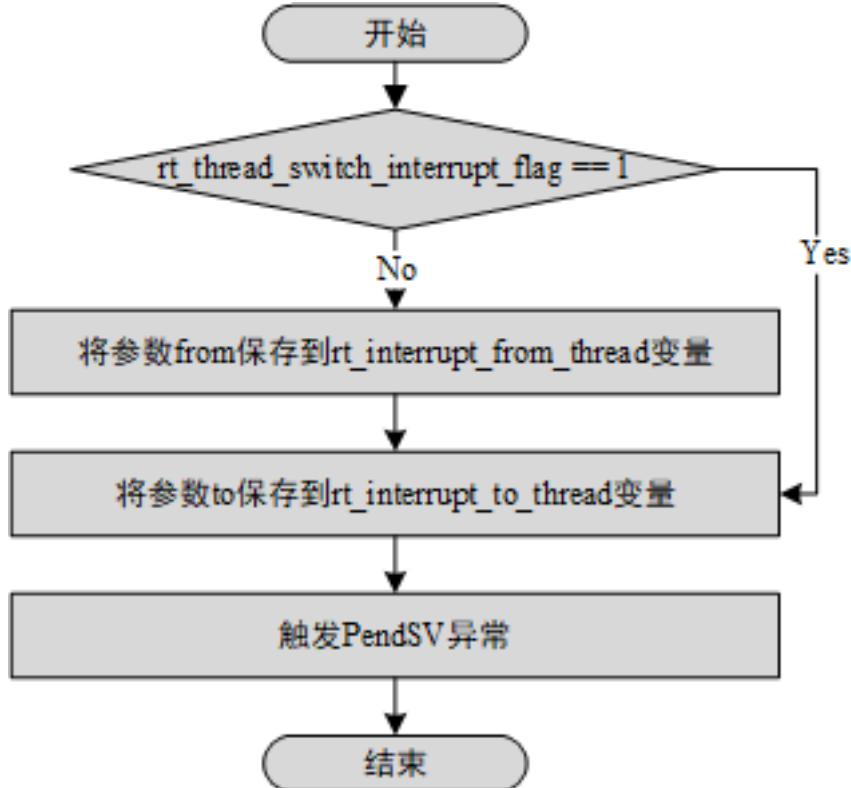
    level = rt_hw_interrupt_disable();
    rt_interrupt_nest++;
    rt_hw_interrupt_enable(level);
}
```

8.2.2.2 用户中断服务程序

在用户中断服务程序（ISR）中，分为两种情况，第一种情况是不进行线程切换，这种情况下用户中断服务程序和中断后续程序运行完毕后退出中断模式，返回被中断的线程。

另一种情况是在中断处理过程中需要进行线程切换，这种情况会调用 `rt_hw_context_switch_interrupt()` 函数进行上下文切换，该函数跟 CPU 架构相关，不同 CPU 架构的实现方式有差异。

在 Cortex-M 架构中，`rt_hw_context_switch_interrupt()` 的函数实现流程如下图所示，它将设置需要切换的线程 `rt_interrupt_to_thread` 变量，然后触发 PendSV 异常（PendSV 异常是专门用来辅助上下文切换的，且被初始化为最低优先级的异常）。PendSV 异常被触发后，不会立即进行 PendSV 异常中断处理程序，因为此时还在中断处理中，只有当中断后续程序运行完毕，真正退出中断处理后，才进入 PendSV 异常中断处理程序。

图 8.7: `rt_hw_context_switch_interrupt()` 函数实现流程

8.2.2.3 中断后续程序

中断后续程序主要完成的工作是：

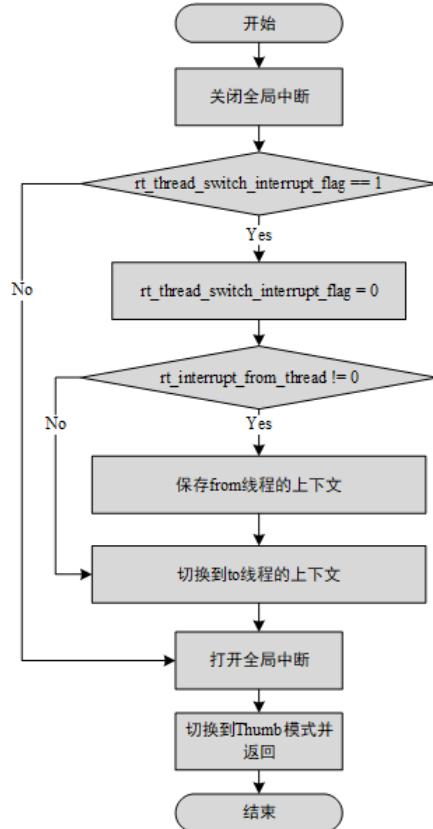
- 1 通知内核离开中断状态，通过调用 `rt_interrupt_leave()` 函数，将全局变量 `rt_interrupt_nest` 减 1，代码如下所示。

```

void rt_interrupt_leave(void)
{
    rt_base_t level;

    level = rt_hw_interrupt_disable();
    rt_interrupt_nest--;
    rt_hw_interrupt_enable(level);
}
  
```

- 2 恢复中断前的 CPU 上下文，如果在中断处理过程中未进行线程切换，那么恢复 `from` 线程的 CPU 上下文，如果在中断中进行了线程切换，那么恢复 `to` 线程的 CPU 上下文。这部分实现跟 CPU 架构相关，不同 CPU 架构的实现方式有差异，在 Cortex-M 架构中实现流程如下图所示。

图 8.8: `rt_hw_context_switch_interrupt()` 函数实现流程

8.2.3 中断嵌套

在允许中断嵌套的情况下，在执行中断服务程序的过程中，如果出现高优先级的中断，当前中断服务程序的执行将被打断，以执行高优先级中断的中断服务程序，当高优先级中断的处理完成后，被打断的中断服务程序才又得到继续执行，如果需要进行线程调度，线程的上下文切换将在所有中断处理程序都运行结束时才发生，如下图所示。

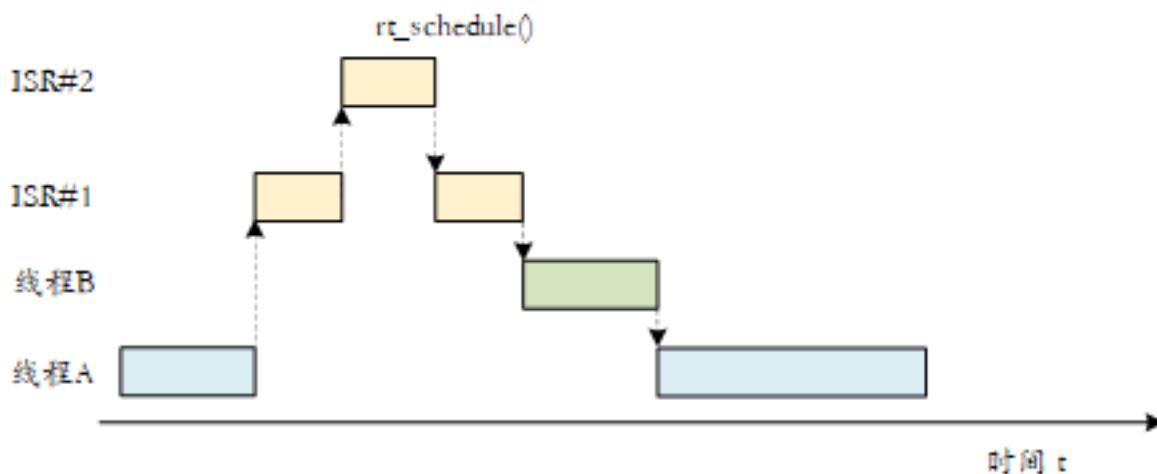


图 8.9: 中断中的线程切换

8.2.4 中断栈

在中断处理过程中，在系统响应中断前，软件代码（或处理器）需要把当前线程的上下文保存下来（通常保存在当前线程的线程栈中），再调用中断服务程序进行中断响应、处理。在进行中断处理时（实质是调用用户的中断服务程序函数），中断处理函数中很可能有自己的局部变量，这些都需要相应的栈空间来保存，所以中断响应依然需要一个栈空间来做为上下文，运行中断处理函数。中断栈可以保存在打断线程的栈中，当从中断中退出时，返回相应的线程继续执行。

中断栈也可以与线程栈完全分离开来，即每次进入中断时，在保存完打断线程上下文后，切换到新的中断栈中独立运行。在中断退出时，再做相应的上下文恢复。使用独立中断栈相对来说更容易实现，并且对于线程栈使用情况也比较容易了解和掌握（否则必须要为中断栈预留空间，如果系统支持中断嵌套，还需要考虑应该为嵌套中断预留多大的空间）。

RT-Thread 采用的方式是提供独立的中断栈，即中断发生时，中断的前期处理程序会将用户的栈指针更换到系统事先留出的中断栈空间中，等中断退出时再恢复用户的栈指针。这样中断就不会占用线程的栈空间，从而提高了内存空间的利用率，且随着线程的增加，这种减少内存占用的效果也越明显。

在 Cortex-M 处理器内核里有两个堆栈指针，一个是主堆栈指针（MSP），是默认的堆栈指针，在运行第一个线程之前和在中断和异常服务程序里使用；另一个是线程堆栈指针（PSP），在线程里使用。在中断和异常服务程序退出时，修改 LR 寄存器的第 2 位的值为 1，线程的 SP 就由 MSP 切换到 PSP。

8.2.5 中断的底半处理

RT-Thread 不对中断服务程序所需要的处理时间做任何假设、限制，但如同其他实时操作系统或非实时操作系统一样，用户需要保证所有的中断服务程序在尽可能短的时间内完成（中断服务程序在系统中相当于拥有最高的优先级，会抢占所有线程优先执行）。这样在发生中断嵌套，或屏蔽了相应中断源的过程中，不会耽误嵌套的其他中断处理过程，或自身中断源的下一次中断信号。

当一个中断发生时，中断服务程序需要取得相应的硬件状态或者数据。如果中断服务程序接下来要对状态或者数据进行简单处理，比如 CPU 时钟中断，中断服务程序只需对一个系统时钟变量进行加一操作，然后就结束中断服务程序。这类中断需要的运行时间往往都比较短。但对于另外一些中断，中断服务程序在取得硬件状态或数据以后，还需要进行一系列更耗时的处理过程，通常需要将该中断分割为两部分，即上半部分（Top Half）和底半部分（Bottom Half）。在上半部分中，取得硬件状态和数据后，打开被屏蔽的中断，给相关线程发送一条通知（可以是 RT-Thread 所提供的信号量、事件、邮箱或消息队列等方式），然后结束中断服务程序；而接下来，相关的线程在接收到通知后，接着对状态或数据进行进一步的处理，这一过程称之为底半处理。

为了详细描述底半处理在 RT-Thread 中的实现，我们以一个虚拟的网络设备接收网络数据包作为范例，如下代码，并假设接收到数据报文后，系统对报文的分析、处理是一个相对耗时的，比外部中断源信号重要性小许多的，而且在不屏蔽中断源信号情况下也能处理的过程。

这个例子的程序创建了一个 nwt 线程，这个线程在启动运行后，将阻塞在 nw_bh_sem 信号上，一旦这个信号量被释放，将执行接下来的 nw_packet_parser 过程，开始 Bottom Half 的事件处理。

```
/*
 * 程序清单：中断底半处理例子
 */

/* 用于唤醒线程的信号量 */
rt_sem_t nw_bh_sem;
```

```

/* 数据读取、分析的线程 */
void demo_nw_thread(void *param)
{
    /* 首先对设备进行必要的初始化工作 */
    device_init_setting();

    /*.. 其他的一些操作..*/

    /* 创建一个 semaphore 来响应 Bottom Half 的事件 */
    nw_bh_sem = rt_sem_create("bh_sem", 0, RT_IPC_FLAG_FIFO);

    while(1)
    {
        /* 最后，让 demo_nw_thread 等待在 nw_bh_sem 上 */
        rt_sem_take(nw_bh_sem, RT_WAITING_FOREVER);

        /* 接收到 semaphore 信号后，开始真正的 Bottom Half 处理过程 */
        nw_packet_parser(packet_buffer);
        nw_packet_process(packet_buffer);
    }
}

int main(void)
{
    rt_thread_t thread;

    /* 创建处理线程 */
    thread = rt_thread_create("nwt", demo_nw_thread, RT_NULL, 1024, 20, 5);

    if (thread != RT_NULL)
        rt_thread_startup(thread);
}

```

接下来让我们来看一下 `demo_nw_isr` 中是如何处理 Top Half，并开启 Bottom Half 的，如下例。

```

void demo_nw_isr(int vector, void *param)
{
    /* 当 network 设备接收到数据后，陷入中断异常，开始执行此 ISR */
    /* 开始 Top Half 部分的处理，如读取硬件设备的状态以判断发生了何种中断 */
    nw_device_status_read();

    /*.. 其他一些数据操作等..*/

    /* 释放 nw_bh_sem，发送信号给 demo_nw_thread，准备开始 Bottom Half */
    rt_sem_release(nw_bh_sem);

    /* 然后退出中断的 Top Half 部分，结束 device 的 ISR */
}

```

从上面例子的两个代码片段可以看出，中断服务程序通过对一个信号量对象的等待和释放，来完成中断 Bottom Half 的起始和终结。由于将中断处理划分为 Top 和 Bottom 两个部分后，使得中断处理过程变为异步过程。这部分系统开销需要用户在使用 RT-Thread 时，必须认真考虑中断服务的处理时间是否大于给 Bottom Half 发送通知并处理的时间。

8.3 RT-Thread 中断管理接口

为了把操作系统和系统底层的异常、中断硬件隔离开来，RT-Thread 把中断和异常封装为一组抽象接口，如下图所示：

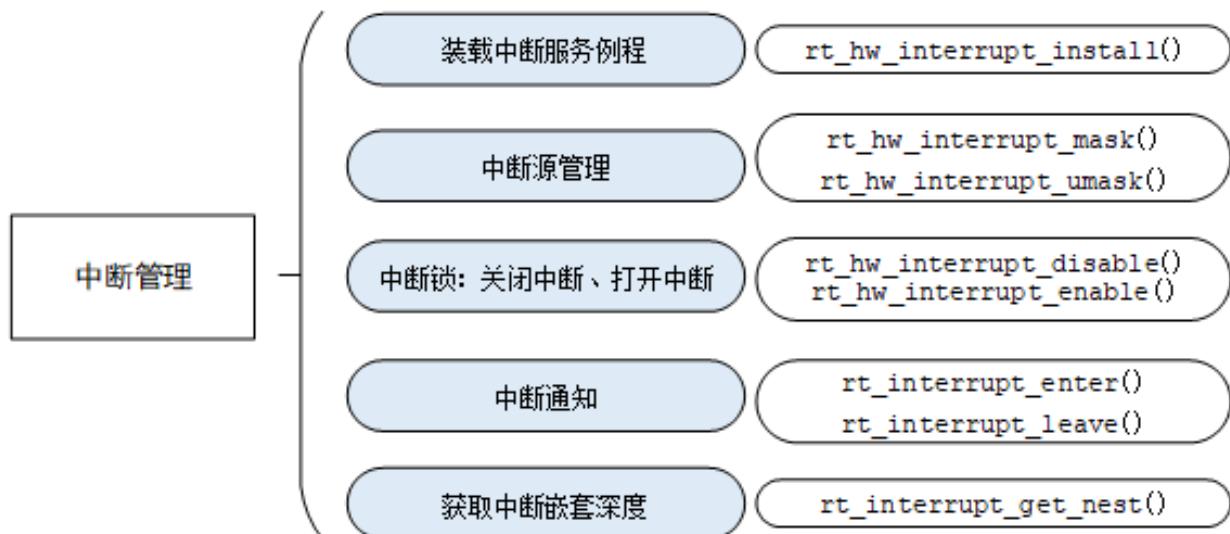


图 8.10：中断相关接口

8.3.1 中断服务程序挂接

系统把用户的中断服务程序 (handler) 和指定的中断号关联起来，可调用如下的接口挂载一个新的中断服务程序：

```
rt_isr_handler_t rt_hw_interrupt_install(int vector,
                                         rt_isr_handler_t handler,
                                         void *param,
                                         char *name);
```

调用 `rt_hw_interrupt_install()` 后，当这个中断源产生中断时，系统将自动调用装载的中断服务程序。下表描述了此函数的输入参数和返回值：

`rt_hw_interrupt_install()` 的输入参数和返回值

参数	描述
vector	vector 是挂载的中断号
handler	新挂载的中断服务程序
param	param 会作为参数传递给中断服务程序

参数	描述
name	中断的名称
返回	—
return	挂载这个中断服务程序之前挂载的中断服务程序的句柄

!!! note “注意事项”这个 API 并不会出现在每一个移植分支中，例如通常 Cortex-M0/M3/M4 的移植分支中就没有这个 API。

中断服务程序是一种需要特别注意的运行环境，它运行在非线程的执行环境下（一般为芯片的一种特殊运行模式（特权模式）），在这个运行环境中不能使用挂起当前线程的操作，因为当前线程并不存在，执行相关的操作会有类似打印提示信息，“Function [abc_func] shall not used in ISR”，含义是不应该在中断服务程序中调用的函数）。

8.3.2 中断源管理

通常在 ISR 准备处理某个中断信号之前，我们需要先屏蔽该中断源，在 ISR 处理完状态或数据以后，及时的打开之前被屏蔽的中断源。

屏蔽中断源可以保证在接下来的处理过程中硬件状态或者数据不会受到干扰，可调用下面这个函数接口：

```
void rt_hw_interrupt_mask(int vector);
```

调用 `rt_hw_interrupt_mask` 函数接口后，相应的中断将会被屏蔽（通常当这个中断触发时，中断状态寄存器会有相应的变化，但并不送到处理器进行处理）。下表描述了此函数的输入参数：

`rt_hw_interrupt_mask()` 的输入参数

参数	描述
vector	要屏蔽的中断号

!!! note “注意事项”这个 API 并不会出现在每一个移植分支中，例如通常 Cortex-M0/M3/M4 的移植分支中就没有这个 API。

为了尽可能的不丢失硬件中断信号，可调用下面的函数接口打开被屏蔽的中断源：

```
void rt_hw_interrupt_umask(int vector);
```

调用 `rt_hw_interrupt_umask` 函数接口后，如果中断（及对应外设）被配置正确时，中断触发后，将送到处理器进行处理。下表描述了此函数的输入参数：

`rt_hw_interrupt_umask()` 的输入参数

参数	描述
vector	要打开屏蔽的中断号

参数	描述
----	----

!!! note “注意事项” 这个 API 并不会出现在每一个移植分支中，例如通常 Cortex-M0/M3/M4 的移植分支中就没有这个 API。

8.3.3 全局中断开关

全局中断开关也称为中断锁，是禁止多线程访问临界区最简单的一种方式，即通过关闭中断的方式，来保证当前线程不会被其他事件打断（因为整个系统已经不再响应那些可以触发线程重新调度的外部事件），也就是当前线程不会被抢占，除非这个线程主动放弃了处理器控制权。当需要关闭整个系统的中断时，可调用下面的函数接口：

```
rt_base_t rt_hw_interrupt_disable(void);
```

下表描述了此函数的返回值：

`rt_hw_interrupt_disable()` 的返回值

返回	描述
中断状态	<code>rt_hw_interrupt_disable</code> 函数运行前的中断状态

恢复中断也称开中断。`rt_hw_interrupt_enable()` 这个函数用于“使能”中断，它恢复了调用 `rt_hw_interrupt_disable()` 函数前的中断状态。如果调用 `rt_hw_interrupt_disable()` 函数前是关中断状态，那么调用此函数后依然是关中断状态。恢复中断往往是和关闭中断成对使用的，调用的函数接口如下：

```
void rt_hw_interrupt_enable(rt_base_t level);
```

下表描述了此函数的输入参数：

`rt_hw_interrupt_enable()` 的输入参数

参数	描述
level	前一次 <code>rt_hw_interrupt_disable</code> 返回的中断状态

1) 使用中断锁来操作临界区的方法可以应用于任何场合，且其他几类同步方式都是依赖于中断锁而实现的，可以说中断锁是最强大的和最高效的同步方法。只是使用中断锁最主要的问题在于，在中断关闭期间系统将不再响应任何中断，也就不能响应外部的事件。所以中断锁对系统的实时性影响非常巨大，当使用不当的时候会导致系统完全无实时性可言（可能导致系统完全偏离要求的时间需求）；而使用得当，则会变成一种快速、高效的同步方式。

例如，为了保证一行代码（例如赋值）的互斥运行，最快速的方法是使用中断锁而不是信号量或互斥量：

```
/* 关闭中断 */
level = rt_hw_interrupt_disable();
a = a + value;
/* 恢复中断 */
rt_hw_interrupt_enable(level);
```

在使用中断锁时，需要确保关闭中断的时间非常短，例如上面代码中的 `a = a + value;` 也可换成另外一种方式，例如使用信号量：

```
/* 获得信号量锁 */
rt_sem_take(sem_lock, RT_WAITING_FOREVER);
a = a + value;
/* 释放信号量锁 */
rt_sem_release(sem_lock);
```

这段代码在 `rt_sem_take`、`rt_sem_release` 的实现中，已经存在使用中断锁保护信号量内部变量的行为，所以对于简单如 `a = a + value;` 的操作，使用中断锁将更为简洁快速。

2) 函数 `rt_base_t rt_hw_interrupt_disable(void)` 和函数 `void rt_hw_interrupt_enable(rt_base_t level)` 一般需要配对使用，从而保证正确的中断状态。

在 RT-Thread 中，开关全局中断的 API 支持多级嵌套使用，简单嵌套中断的代码如下代码所示：

简单嵌套中断使用

```
#include <rthw.h>

void global_interrupt_demo(void)
{
    rt_base_t level0;
    rt_base_t level1;

    /* 第一次关闭全局中断，关闭之前的全局中断状态可能是打开的，也可能是关闭的 */
    level0 = rt_hw_interrupt_disable();
    /* 第二次关闭全局中断，关闭之前的全局中断是关闭的，关闭之后全局中断还是关闭的 */
    level1 = rt_hw_interrupt_disable();

    do_something();

    /* 恢复全局中断到第二次关闭之前的状态，所以本次 enable 之后全局中断还是关闭的 */
    rt_hw_interrupt_enable(level1);
    /* 恢复全局中断到第一次关闭之前的状态，这时候的全局中断状态可能是打开的，也可能是关闭的 */
    rt_hw_interrupt_enable(level0);
}
```

这个特性可以给代码的开发带来很大的便利。例如在某个函数里关闭了中断，然后调用某些子函数，再打开中断。这些子函数里面也可能存在开关中断的代码。由于全局中断的 API 支持嵌套使用，用户无需为这些代码做特殊处理。

8.3.4 中断通知

当整个系统被中断打断，进入中断处理函数时，需要通知内核当前已经进入到中断状态。针对这种情况，可通过以下接口：

```
void rt_interrupt_enter(void);
void rt_interrupt_leave(void);
```

这两个接口分别用在中断前导程序和中断后续程序中，均会对 `rt_interrupt_nest`（中断嵌套深度）的值进行修改：

每当进入中断时，可以调用 `rt_interrupt_enter()` 函数，用于通知内核，当前已经进入了中断状态，并增加中断嵌套深度（执行 `rt_interrupt_nest++`）；

每当退出中断时，可以调用 `rt_interrupt_leave()` 函数，用于通知内核，当前已经离开了中断状态，并减少中断嵌套深度（执行 `rt_interrupt_nest--`）。注意不要在应用程序中调用这两个接口函数。

使用 `rt_interrupt_enter/leave()` 的作用是，在中断服务程序中，如果调用了内核相关的函数（如释放信号量等操作），则可以通过判断当前中断状态，让内核及时调整相应的行为。例如：在中断中释放了一个信号量，唤醒了某线程，但通过判断发现当前系统处于中断上下文环境中，那么在进行线程切换时应该采取中断中线程切换的策略，而不是立即进行切换。

但如果中断服务程序不会调用内核相关的函数（释放信号量等操作），这个时候，也可以不调用 `rt_interrupt_enter/leave()` 函数。

在上层应用中，在内核需要知道当前已经进入到中断状态或当前嵌套的中断深度时，可调用 `rt_interrupt_get_nest()` 接口，它会返回 `rt_interrupt_nest`。如下：

```
rt_uint8_t rt_interrupt_get_nest(void);
```

下表描述了 `rt_interrupt_get_nest()` 的返回值

返回	描述
0	当前系统不处于中断上下文环境中
1	当前系统处于中断上下文环境中
大于 1	当前中断嵌套层次

8.4 中断与轮询

当驱动外设工作时，其编程模式到底采用中断模式触发还是轮询模式触发往往是驱动开发人员首先要考虑的问题，并且这个问题在实时操作系统与分时操作系统中差异还非常大。因为轮询模式本身采用顺序执行的方式：查询到相应的事件然后进行对应的处理。所以轮询模式从实现上来说，相对简单清晰。例如往串口中写入数据，仅当串口控制器写完一个数据时，程序代码才写入下一个数据（否则这个数据丢弃掉）。相应的代码可以是这样的：

```
/* 轮询模式向串口写入数据 */
while (size)
```

```

{
    /* 判断 UART 外设中数据是否发送完毕 */
    while (!(uart->uart_device->SR & USART_FLAG_TXE));
    /* 当所有数据发送完毕后，才发送下一个数据 */
    uart->uart_device->DR = (*ptr & 0xFF);

    ++ptr; --size;
}

```

在实时系统中轮询模式可能会出现非常大问题，因为在实时操作系统中，当一个程序持续地执行时（轮询时），它所在的线程会一直运行，比它优先级低的线程都不会得到运行。而分时系统中，这点恰恰相反，几乎没有优先级之分，可以在一个时间片运行这个程序，然后在另外一段时间片上运行另外一段程序。

所以通常情况下，实时系统中更多采用的是中断模式来驱动外设。当数据达到时，由中断唤醒相关的处理线程，再继续进行后续的动作。例如一些携带 FIFO（包含一定数据量的先进先出队列）的串口外设，其写入过程可以是这样的，如下图所示：

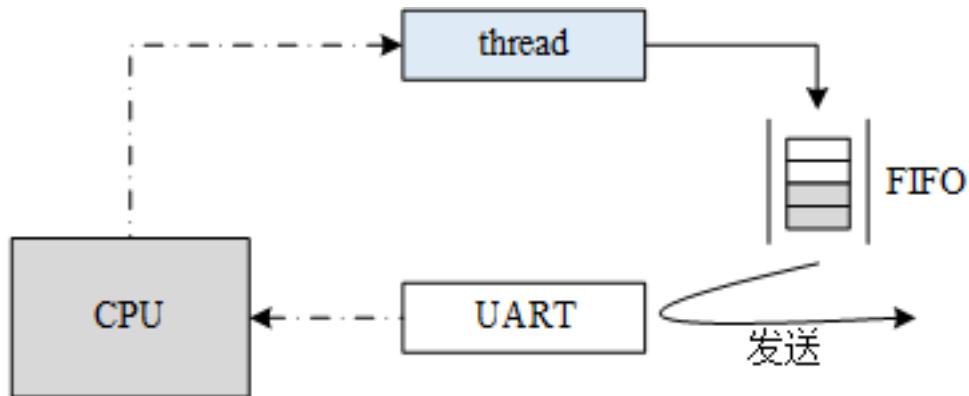


图 8.11：中断模式驱动外设

线程先向串口的 FIFO 中写入数据，当 FIFO 满时，线程主动挂起。串口控制器持续地从 FIFO 中取出数据并以配置的波特率（例如 115200bps）发送出去。当 FIFO 中所有数据都发送完成时，将向处理器触发一个中断；当中断服务程序得到执行时，可以唤醒这个线程。这里举例的是 FIFO 类型的设备，在现实中也有 DMA 类型的设备，原理类似。

对于低速设备来说，运用这种模式非常好，因为在串口外设把 FIFO 中的数据发送出去前，处理器可以运行其他的线程，这样就提高了系统的整体运行效率（甚至对于分时系统来说，这样的设计也是非常必要）。但是对于一些高速设备，例如传输速度达到 10Mbps 的时候，假设一次发送的数据量是 32 字节，我们可以计算出发送这样一段数据量需要的时间是： $(32 \times 8) \times 1/10\text{Mbps} = 25\mu\text{s}$ 。当数据需要持续传输时，系统将在 $25\mu\text{s}$ 后触发一个中断以唤醒上层线程继续下次传递。假设系统的线程切换时间是 $8\mu\text{s}$ （通常实时操作系统的线程上下文切换时间只有几个 μs ），那么当整个系统运行时，对于数据带宽利用率将只有 $25/(25+8) = 75.8\%$ 。但是采用轮询模式，数据带宽的利用率则可能达到 100%。这个也是大家普遍认为实时系统中数据吞吐量不足的缘故，系统开销消耗在了线程切换上（有些实时系统甚至会如本章前面说的，采用底半处理，分级的中断处理方式，相当于又拉长中断到发送线程的时间开销，效率会更进一步下降）。

通过上述的计算过程，我们可以看出其中的一些关键因素：发送数据量越小，发送速度越快，对于数据吞吐量的影响也将越大。归根结底，取决于系统中产生中断的频度如何。当一个实时系统想要提升数据吞吐量时，可以考虑的几种方式：

- 1) 增加每次数据量发送的长度，每次尽量让外设尽量多地发送数据；

2) 必要情况下更改中断模式为轮询模式。同时为了解决轮询方式一直抢占处理机，其他低优先级线程得不到运行的情况，可以把轮询线程的优先级适当降低。

8.5 全局中断开关使用示例

这是一个中断的应用例程：在多线程访问同一个变量时，使用开关全局中断对该变量进行保护，如下代码所示：

使用开关中断进行全局变量的访问

```
#include <rthw.h>
#include <rtthread.h>

#define THREAD_PRIORITY      20
#define THREAD_STACK_SIZE    512
#define THREAD_TIMESLICE     5

/* 同时访问的全局变量 */
static rt_uint32_t cnt;

void thread_entry(void *parameter)
{
    rt_uint32_t no;
    rt_uint32_t level;

    no = (rt_uint32_t) parameter;
    while (1)
    {
        /* 关闭全局中断 */
        level = rt_hw_interrupt_disable();
        cnt += no;
        /* 恢复全局中断 */
        rt_hw_interrupt_enable(level);

        rt_kprintf("protect thread[%d]'s counter is %d\n", no, cnt);
        rt_thread_mdelay(no * 10);
    }
}

/* 用户应用程序入口 */
int interrupt_sample(void)
{
    rt_thread_t thread;

    /* 创建 t1 线程 */
    thread = rt_thread_create("thread1", thread_entry, (void *)10,
                             THREAD_STACK_SIZE,
                             THREAD_PRIORITY, THREAD_TIMESLICE);
    if (thread != RT_NULL)
        rt_thread_startup(thread);
```

```
/* 创建 t2 线程 */
thread = rt_thread_create("thread2", thread_entry, (void *)20,
                          THREAD_STACK_SIZE,
                          THREAD_PRIORITY, THREAD_TIMESLICE);
if (thread != RT_NULL)
    rt_thread_startup(thread);

return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(interrupt_sample, interrupt sample);
```

仿真运行结果如下：

```
\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Aug 27 2018
2006 - 2018 Copyright by rt-thread team
msh >interrupt_sample
msh >protect thread[10]'s counter is 10
protect thread[20]'s counter is 30
protect thread[10]'s counter is 40
protect thread[20]'s counter is 60
protect thread[10]'s counter is 70
protect thread[10]'s counter is 80
protect thread[20]'s counter is 100
protect thread[10]'s counter is 110
protect thread[10]'s counter is 120
protect thread[20]'s counter is 140
...
...
```

!!! note “注意事项”由于关闭全局中断会导致整个系统不能响应中断，所以在使用关闭全局中断做为互斥访问临界区的手段时，必须需要保证关闭全局中断的时间非常短，例如运行数条机器指令的时间。

第 9 章

内核移植

经过前面内核章节的学习，大家对 RT-Thread 也有了不少的了解，但是如何将 RT-Thread 内核移植到不同的硬件平台上，很多人还不一定熟悉。内核移植就是指将 RT-Thread 内核在不同的芯片架构、不同的板卡上运行起来，能够具备线程管理和调度，内存管理，线程间同步和通信、定时器管理等功能。移植可分为 CPU 架构移植和 BSP（Board support package，板级支持包）移植两部分。

本章将展开介绍 CPU 架构移植和 BSP 移植，CPU 架构移植部分会结合 Cortex-M CPU 架构进行介绍，因此有必要回顾下上一章《中断管理》介绍的“Cortex-M CPU 架构基础”的内容，本章最后以实际移植到一个开发板的示例展示 RT-Thread 内核移植的完整过程，读完本章，我们将了解如何完成 RT-Thread 的内核移植。

9.1 CPU 架构移植

在嵌入式领域有多种不同 CPU 架构，例如 Cortex-M、ARM920T、MIPS32、RISC-V 等等。为了使 RT-Thread 能够在不同 CPU 架构的芯片上运行，RT-Thread 提供了一个 libcpu 抽象层来适配不同的 CPU 架构。libcpu 层向上对内核提供统一的接口，包括全局中断的开关，线程栈的初始化，上下文切换等。

RT-Thread 的 libcpu 抽象层向下提供了一套统一的 CPU 架构移植接口，这部分接口包含了全局中断开关函数、线程上下文切换函数、时钟节拍的配置和中断函数、Cache 等等内容。下表是 CPU 架构移植需要实现的接口和变量。

libcpu 移植相关 API

函数和变量	描述
<code>rt_base_t rt_hw_interrupt_disable(void);</code>	关闭全局中断
<code>void rt_hw_interrupt_enable(rt_base_t level);</code>	打开全局中断
<code>rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter, rt_uint8_t *stack_addr, void *texit);</code>	线程栈的初始化，内核在线程创建和线程初始化里面会调用这个函数
<code>void rt_hw_context_switch_to(rt_uint32 to);</code>	没有来源线程的上下文切换，在调度器启动第一个线程的时候调用，以及在 <code>signal</code> 里面会调用

函数和变量	描述
void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);	从 from 线程切换到 to 线程, 用于线程和线程之间的切换
void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);	从 from 线程切换到 to 线程, 用于中断里面进行切换的时候使用
rt_uint32_t rt_thread_switch_interrupt_flag;	表示需要在中断里进行切换的标志
rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;	在线程进行上下文切换时候, 用来保存 from 和 to 线程

9.1.1 实现全局中断开关

无论内核代码还是用户的代码, 都可能存在一些变量, 需要在多个线程或者中断里面使用, 如果没有相应的保护机制, 那就可能导致临界区问题。RT-Thread 里为了解决这个问题, 提供了一系列的线程间同步和通信机制来解决。但是这些机制都需要用到 libcpu 里提供的全局中断开关函数。它们分别是:

```
/* 关闭全局中断 */
rt_base_t rt_hw_interrupt_disable(void);

/* 打开全局中断 */
void rt_hw_interrupt_enable(rt_base_t level);
```

下面介绍在 Cortex-M 架构上如何实现这两个函数, 前文中曾提到过, Cortex-M 为了快速开关中断, 实现了 CPS 指令, 可以用在此处。

```
CPSID I ;PRIMASK=1, ; 关中断
CPSIE I ;PRIMASK=0, ; 开中断
```

9.1.1.1 关闭全局中断

在 `rt_hw_interrupt_disable()` 函数里面需要依序完成的功能是:

- 1) . 保存当前的全局中断状态, 并把状态作为函数的返回值。
- 2) . 关闭全局中断。

基于 MDK, 在 Cortex-M 内核上实现关闭全局中断, 如下代码所示:

关闭全局中断

```
/*
; * rt_base_t rt_hw_interrupt_disable(void);
; */
rt_hw_interrupt_disable PROC ;PROC 伪指令定义函数
    EXPORT rt_hw_interrupt_disable ;EXPORT 输出定义的函数, 类似于 C 语言 extern
    MRS r0, PRIMASK ;读取 PRIMASK 寄存器的值到 r0 寄存器
    CPSID I ;关闭全局中断
    BX LR ;函数返回
```

ENDP

;ENDP 函数结束

上面的代码首先是使用 MRS 指令将 PRIMASK 寄存器的值保存到 r0 寄存器里，然后使用“CPSID I”指令关闭全局中断，最后使用 BX 指令返回。r0 存储的数据就是函数的返回值。中断可以发生在“MRS r0, PRIMASK”指令和“CPSID I”之间，这并不会导致全局中断状态的错乱。

关于寄存器在函数调用的时候和在中断处理程序里是如何管理的，不同的 CPU 架构有不同的约定。在 ARM 官方手册《Procedure Call Standard for the ARM ® Architecture》里可以找到关于 Cortex-M 的更详细的介绍寄存器使用的约定。

9.1.1.2 打开全局中断

在 rt_hw_interrupt_enable(rt_base_t level) 里，将变量 level 作为需要恢复的状态，覆盖芯片的全局中断状态。

基于 MDK，在 Cortex-M 内核上的实现打开全局中断，如下代码所示：

打开全局中断

```
;/*
; * void rt_hw_interrupt_enable(rt_base_t level);
; */

rt_hw_interrupt_enable PROC      ; PROC 伪指令定义函数
    EXPORT rt_hw_interrupt_enable ; EXPORT 输出定义的函数，类似于 C 语言 extern
    MSR    PRIMASK, r0          ; 将 r0 寄存器的值写入到 PRIMASK 寄存器
    BX     LR                  ; 函数返回
    ENDP                           ; ENDP 函数结束
```

上面的代码首先是使用 MSR 指令将 r0 的值寄存器写入到 PRIMASK 寄存器，从而恢复之前的中断状态。

9.1.2 实现线程栈初始化

在动态创建线程和初始化线程的时候，会使用到内部的线程初始化函数 _rt_thread_init()，_rt_thread_init() 函数会调用栈初始化函数 rt_hw_stack_init()，在栈初始化函数里会手动构造一个上下文内容，这个上下文内容将被作为每个线程第一次执行的初始值。上下文在栈里的排布如下图所示：



图 9.1: 栈里的上下文信息

下代码是栈初始化的代码：

在栈里构建上下文

```
rt_uint8_t *rt_hw_stack_init(void          *tentry,
                             void          *parameter,
                             rt_uint8_t   *stack_addr,
                             void          *texit)
{
    struct stack_frame *stack_frame;
    rt_uint8_t         *stk;
    unsigned long      i;

    /* 对传入的栈指针做对齐处理 */
    stk = stack_addr + sizeof(rt_uint32_t);
    stk = (rt_uint8_t *)RT_ALIGN_DOWN((rt_uint32_t)stk, 8);
    stk -= sizeof(struct stack_frame);

    /* 得到上下文的栈帧的指针 */
    stack_frame = (struct stack_frame *)stk;

    /* 把所有寄存器的默认值设置为 0xdeadbeef */
    /* ... (rest of the code) ... */
}
```

```

for (i = 0; i < sizeof(struct stack_frame) / sizeof(rt_uint32_t); i++)
{
    ((rt_uint32_t *)stack_frame)[i] = 0xdeadbeef;
}

/* 根据 ARM APCS 调用标准, 将第一个参数保存在 r0 寄存器 */
stack_frame->exception_stack_frame.r0 = (unsigned long)parameter;
/* 将剩下的参数寄存器都设置为 0 */
stack_frame->exception_stack_frame.r1 = 0; /* r1 寄存器 */
stack_frame->exception_stack_frame.r2 = 0; /* r2 寄存器 */
stack_frame->exception_stack_frame.r3 = 0; /* r3 寄存器 */
/* 将 IP(Intra-Procedure-call scratch register.) 设置为 0 */
stack_frame->exception_stack_frame.r12 = 0; /* r12 寄存器 */
/* 将线程退出函数的地址保存在 lr 寄存器 */
stack_frame->exception_stack_frame.lr = (unsigned long)texit;
/* 将线程入口函数的地址保存在 pc 寄存器 */
stack_frame->exception_stack_frame.pc = (unsigned long)tentry;
/* 设置 psr 的值为 0x01000000L, 表示默认切换过去是 Thumb 模式 */
stack_frame->exception_stack_frame.psr = 0x01000000L;

/* 返回当前线程的栈地址 */
return stk;
}

```

9.1.3 实现上下文切换

在不同的 CPU 架构里, 线程之间的上下文切换和中断到线程的上下文切换, 上下文的寄存器部分可能是有差异的, 也可能是一样的。在 Cortex-M 里面上下文切换都是统一使用 PendSV 异常来完成, 切换部分并没有差异。但是为了能适应不同的 CPU 架构, RT-Thread 的 libcpu 抽象层还是需要实现三个线程切换相关的函数:

- 1) `rt_hw_context_switch_to()`: 没有来源线程, 切换到目标线程, 在调度器启动第一个线程的时候被调用。
- 2) `rt_hw_context_switch()`: 在线程环境下, 从当前线程切换到目标线程。
- 3) `rt_hw_context_switch_interrupt()`: 在中断环境下, 从当前线程切换到目标线程。

在线程环境下进行切换和在中断环境进行切换是存在差异的。线程环境下, 如果调用 `rt_hw_context_switch()` 函数, 那么可以马上进行上下文切换; 而在中断环境下, 需要等待中断处理函数完成之后才能进行切换。

由于这种差异, 在 ARM9 等平台, `rt_hw_context_switch()` 和 `rt_hw_context_switch_interrupt()` 的实现并不一样。在中断处理程序里如果触发了线程的调度, 调度函数里会调用 `rt_hw_context_switch_interrupt()` 触发上下文切换。中断处理程序里处理完中断事务之后, 中断退出之前, 检查 `rt_thread_switch_interrupt_flag` 变量, 如果该变量的值为 1, 就根据 `rt_interrupt_from_thread` 变量和 `rt_interrupt_to_thread` 变量, 完成线程的上下文切换。

在 Cortex-M 处理器架构里, 基于自动部分压栈和 PendSV 的特性, 上下文切换可以实现地更加简洁。

线程之间的上下文切换, 如下图表示:

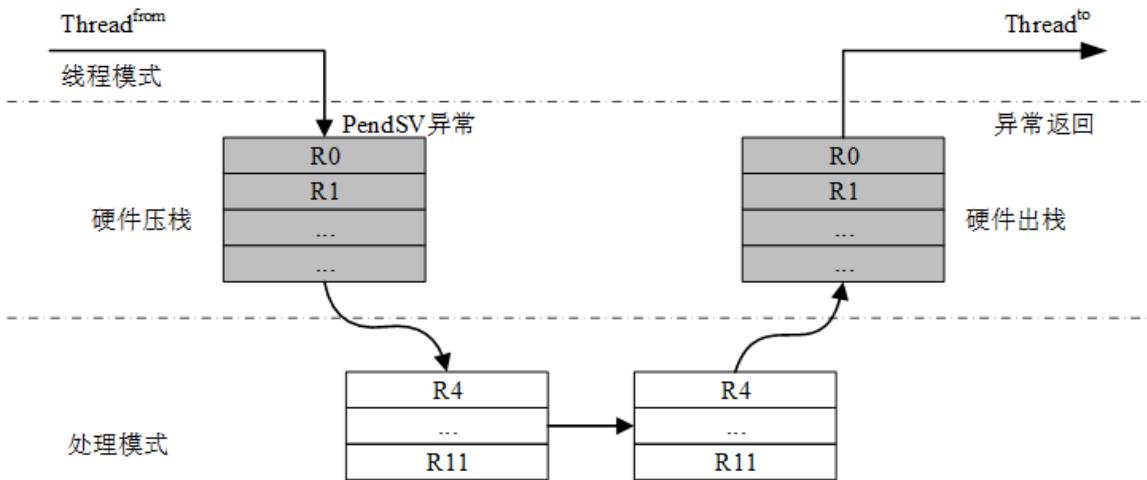


图 9.2: 线程之间的上下文切换

硬件在进入 PendSV 中断之前自动保存了 from 线程的 PSR、PC、LR、R12、R3-R0 寄存器，然后 PendSV 里保存 from 线程的 R11~R4 寄存器，以及恢复 to 线程的 R4~R11 寄存器，最后硬件在退出 PendSV 中断之后，自动恢复 to 线程的 R0~R3、R12、LR、PC、PSR 寄存器。

中断到线程的上下文切换可以用下图表示：

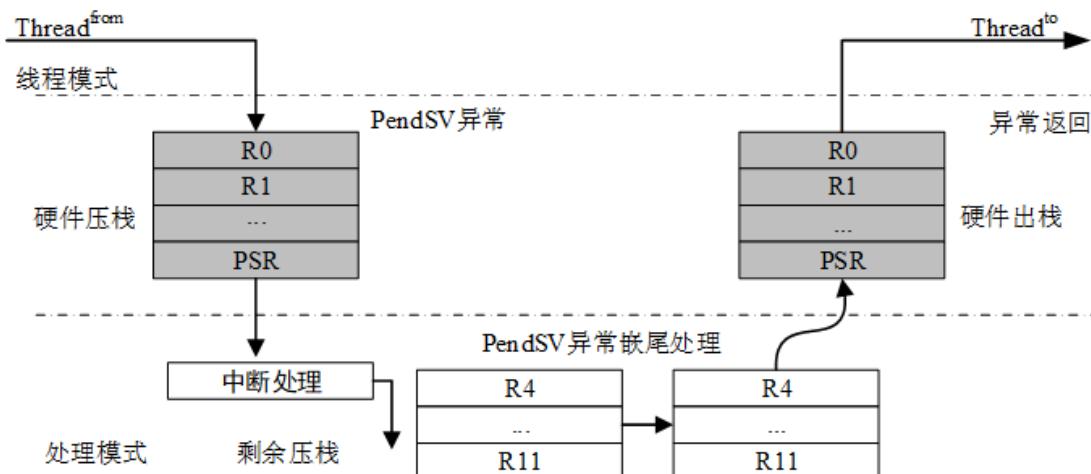


图 9.3: 中断到线程的切换

硬件在进入中断之前自动保存了 from 线程的 PSR、PC、LR、R12、R3-R0 寄存器，然后触发了 PendSV 异常。在 PendSV 异常处理函数里保存 from 线程的 R11~R4 寄存器，以及恢复 to 线程的 R4~R11 寄存器，最后硬件在退出 PendSV 中断之后，自动恢复 to 线程的 R0~R3、R12、PSR、PC、LR 寄存器。

显然，在 Cortex-M 内核里 `rt_hw_context_switch()` 和 `rt_hw_context_switch_interrupt()` 功能一致，都是在 PendSV 里完成剩余上下文的保存和回复。所以我们仅仅需要实现一份代码，简化移植的工作。

9.1.3.1 实现 rt_hw_context_switch_to()

`rt_hw_context_switch_to()` 只有目标线程，没有来源线程。这个函数里实现切换到指定线程的功能，下图是流程图：

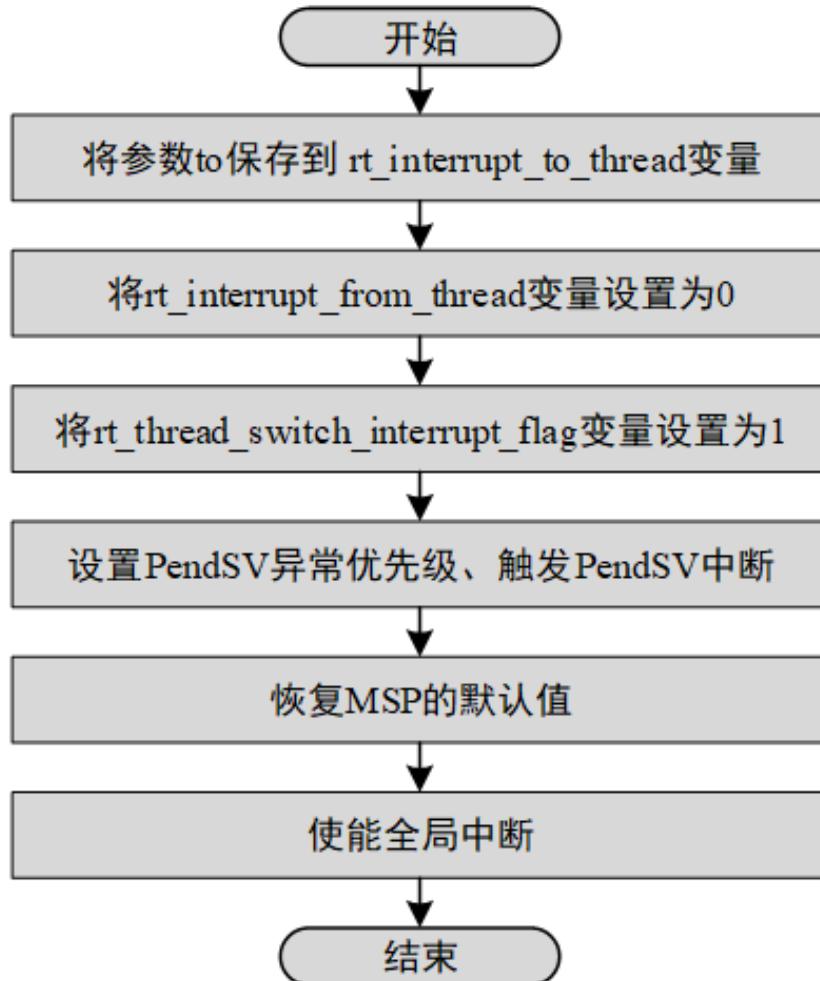


图 9.4: `rt_hw_context_switch_to()` 流程图

在 Cortex-M3 内核上的 `rt_hw_context_switch_to()` 实现（基于 MDK），如下代码所示：

MDK 版 `rt_hw_context_switch_to()` 实现

```

;/*
; * void rt_hw_context_switch_to(rt_uint32 to);
; * r0 --> to
; * this function is used to perform the first thread switch
; */
rt_hw_context_switch_to    PROC
    EXPORT rt_hw_context_switch_to
    ; r0 的值是一个指针，该指针指向 to 线程的线程控制块的 SP 成员
    ; 将 r0 寄存器的值保存到 rt_interrupt_to_thread 变量里
    LDR    r1, =rt_interrupt_to_thread
  
```

```
STR      r0, [r1]

; 设置 from 线程为空，表示不需要从保存 from 的上下文
LDR      r1, =rt_interrupt_from_thread
MOV      r0, #0x0
STR      r0, [r1]

; 设置标志为 1，表示需要切换，这个变量将在 PendSV 异常处理函数里切换的时被清零
LDR      r1, =rt_thread_switch_interrupt_flag
MOV      r0, #1
STR      r0, [r1]

; 设置 PendSV 异常优先级为最低优先级
LDR      r0, =NVIC_SYSPRI2
LDR      r1, =NVIC_PENDSV_PRI
LDR.W   r2, [r0,#0x00]          ; read
ORR     r1,r1,r2              ; modify
STR      r1, [r0]                ; write-back

; 触发 PendSV 异常（将执行 PendSV 异常处理程序）
LDR      r0, =NVIC_INT_CTRL
LDR      r1, =NVIC_PENDSVSET
STR      r1, [r0]

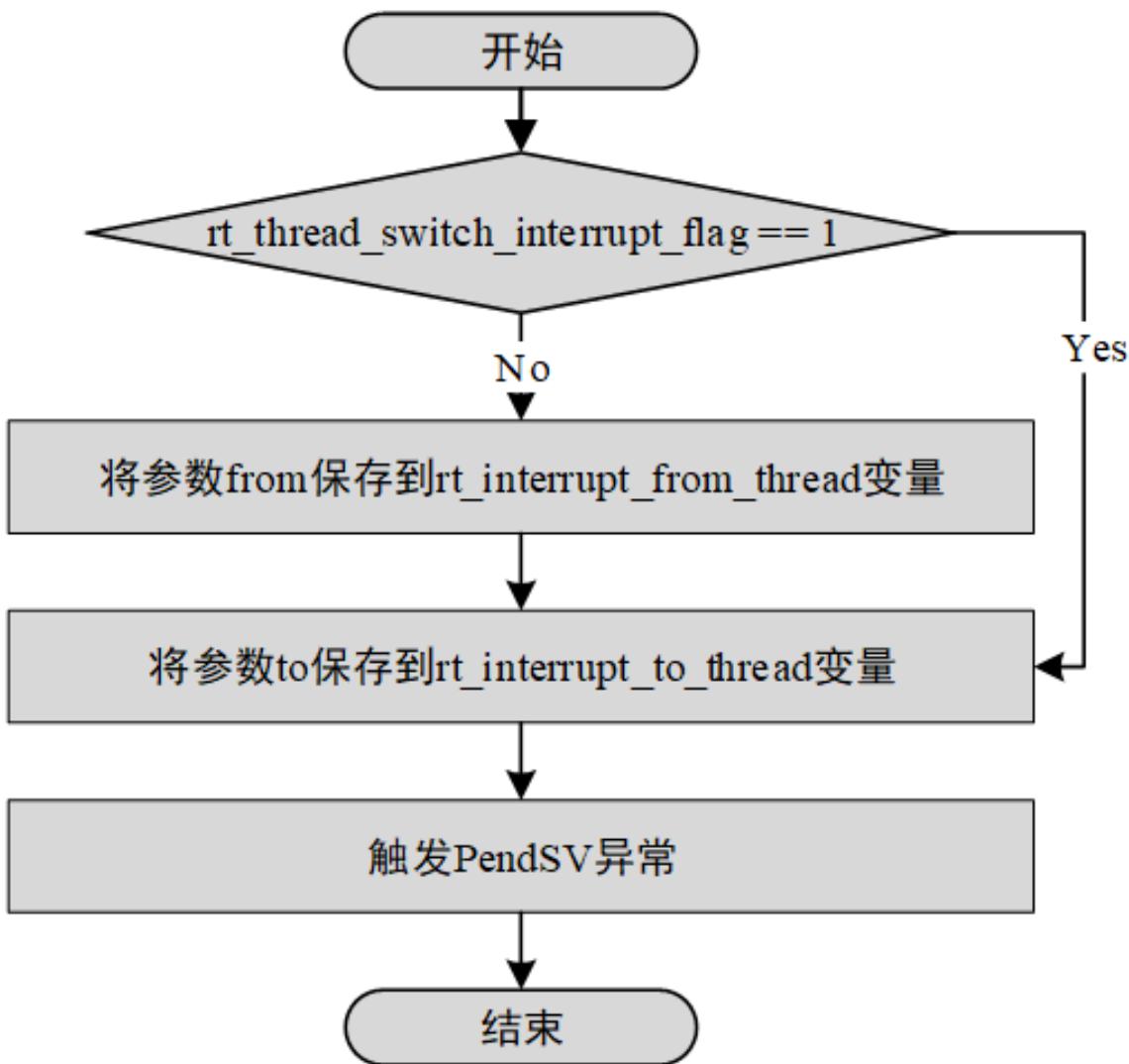
; 放弃芯片启动到第一次上下文切换之前的栈内容，将 MSP 设置启动时的值
LDR      r0, =SCB_VTOR
LDR      r0, [r0]
LDR      r0, [r0]
MSR      msp, r0

; 使能全局中断和全局异常，使能之后将进入 PendSV 异常处理函数
CPSIE   F
CPSIE   I

; 不会执行到这里
ENDP
```

9.1.3.2 实现 rt_hw_context_switch() / rt_hw_context_switch_interrupt()

函数 `rt_hw_context_switch()` 和函数 `rt_hw_context_switch_interrupt()` 都有两个参数，分别是 `from` 线程和 `to` 线程。它们实现从 `from` 线程切换到 `to` 线程的功能。下图是具体的流程图：

图 9.5: `rt_hw_context_switch()`/`rt_hw_context_switch_interrupt()` 流程图

在 Cortex-M3 内核上的 `rt_hw_context_switch()` 和 `rt_hw_context_switch_interrupt()` 实现（基于 MDK），如下代码所示：

`rt_hw_context_switch()`/`rt_hw_context_switch_interrupt()` 实现

```

; /*
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; * r0 --> from
; * r1 --> to
; */
rt_hw_context_switch_interrupt
  EXPORT rt_hw_context_switch_interrupt
rt_hw_context_switch      PROC
  EXPORT rt_hw_context_switch

  ; 检查 rt_thread_switch_interrupt_flag 变量是否为 1
  ; 如果变量为 1 就跳过更新 from 线程的内容

```

```
LDR    r2, =rt_thread_switch_interrupt_flag
LDR    r3, [r2]
CMP    r3, #1
BEQ    _reswitch
; 设置 rt_thread_switch_interrupt_flag 变量为 1
MOV    r3, #1
STR    r3, [r2]

; 从参数 r0 里更新 rt_interrupt_from_thread 变量
LDR    r2, =rt_interrupt_from_thread
STR    r0, [r2]

_reswitch
; 从参数 r1 里更新 rt_interrupt_to_thread 变量
LDR    r2, =rt_interrupt_to_thread
STR    r1, [r2]

; 触发 PendSV 异常，将进入 PendSV 异常处理函数里完成上下文切换
LDR    r0, =NVIC_INT_CTRL
LDR    r1, =NVIC_PENDSVSET
STR    r1, [r0]
BX    LR
```

9.1.3.3 实现 PendSV 中断

在 Cortex-M3 里，PendSV 中断处理函数是 PendSV_Handler()。在 PendSV_Handler() 里完成线程切换的实际工作，下图是具体的流程图：

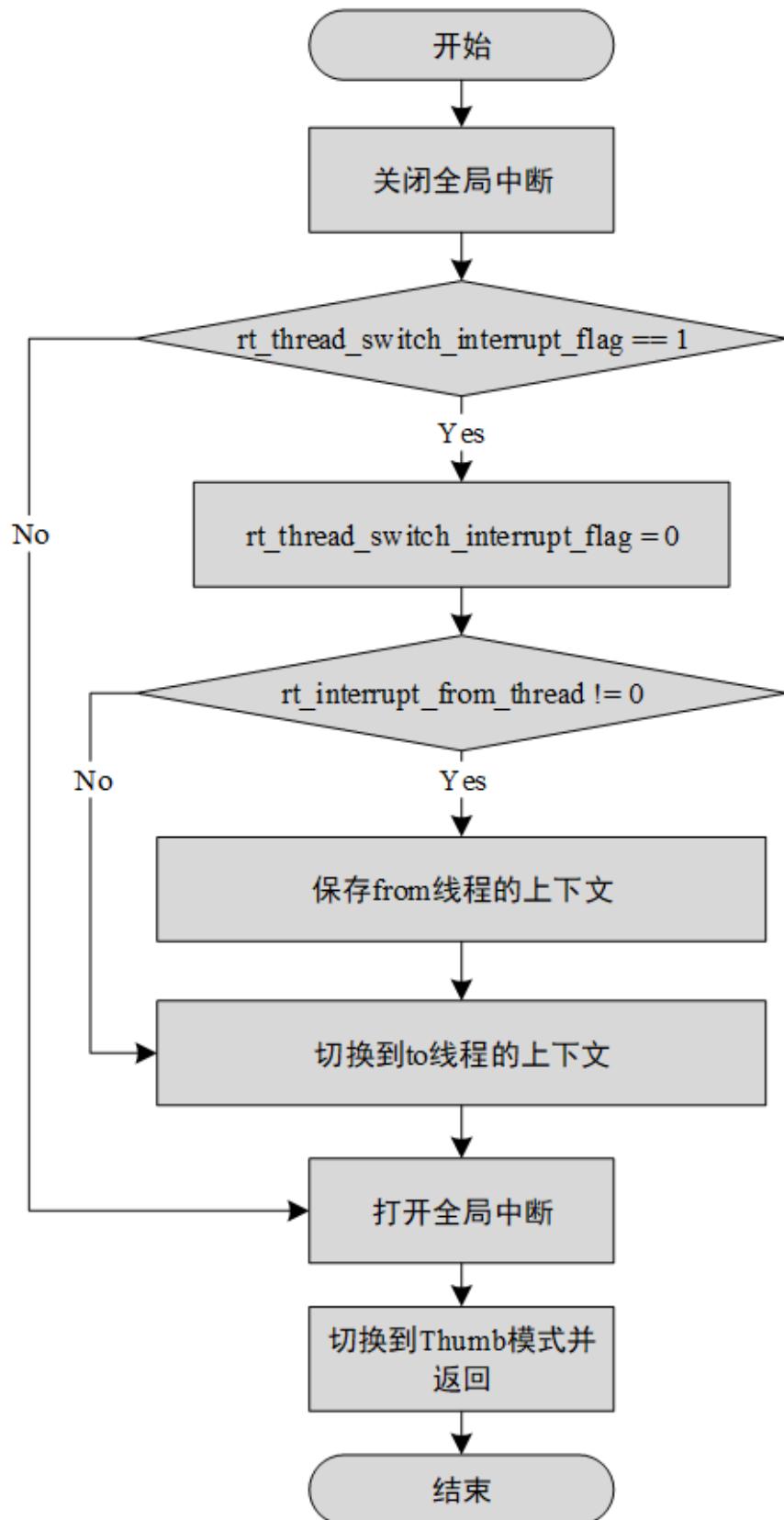


图 9.6: PendSV 中断处理

如下代码是 PendSV_Handler 实现:

```

; r0 --> switch from thread stack
; r1 --> switch to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from] stack
PendSV_Handler PROC
    EXPORT PendSV_Handler

    ; 关闭全局中断
    MRS      r2, PRIMASK
    CPSID    I

    ; 检查 rt_thread_switch_interrupt_flag 变量是否为 0
    ; 如果为零就跳转到 pendsv_exit
    LDR      r0, =rt_thread_switch_interrupt_flag
    LDR      r1, [r0]
    CBZ      r1, pendsv_exit          ; pendsv already handled

    ; 清零 rt_thread_switch_interrupt_flag 变量
    MOV      r1, #0x00
    STR      r1, [r0]

    ; 检查 rt_thread_switch_interrupt_flag 变量
    ; 如果为 0，就不进行 from 线程的上下文保存
    LDR      r0, =rt_interrupt_from_thread
    LDR      r1, [r0]
    CBZ      r1, switch_to_thread

    ; 保存 from 线程的上下文
    MRS      r1, psp                  ; 获取 from 线程的栈指针
    STMFD   r1!, {r4 - r11}           ; 将 r4~r11 保存到线程的栈里
    LDR      r0, [r0]
    STR      r1, [r0]                 ; 更新线程的控制块的 SP 指针

switch_to_thread
    LDR      r1, =rt_interrupt_to_thread
    LDR      r1, [r1]
    LDR      r1, [r1]                 ; 获取 to 线程的栈指针

    LDMFD   r1!, {r4 - r11}           ; 从 to 线程的栈里恢复 to 线程的寄存器值
    MSR      psp, r1                 ; 更新 r1 的值到 psp

pendsv_exit
    ; 恢复全局中断状态
    MSR      PRIMASK, r2

    ; 修改 lr 寄存器的 bit2，确保进程使用 PSP 堆栈指针
    ORR      lr, lr, #0x04
    ; 退出中断函数
    BX      lr
ENDP

```

9.1.4 实现时钟节拍

有了开关全局中断和上下文切换功能的基础，RTOS 就可以进行线程的创建、运行、调度等功能了。有了时钟节拍支持，RT-Thread 可以实现对相同优先级的线程采用时间片轮转的方式来调度，实现定时器功能，实现 `rt_thread_delay()` 延时函数等等。

`libcpu` 的移植需要完成的工作，就是确保 `rt_tick_increase()` 函数会在时钟节拍的中断里被周期性的调用，调用周期取决于 `rtconfig.h` 的宏 `RT_TICK_PER_SECOND` 的值。

在 Cortex M 中，实现 `SysTick` 的中断处理函数即可实现时钟节拍功能。

```
void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}
```

9.2 BSP 移植

相同的 CPU 架构在实际项目中，不同的板卡上可能使用相同的 CPU 架构，搭载不同的外设资源，完成不同的产品，所以我们也需要针对板卡做适配工作。RT-Thread 提供了 BSP 抽象层来适配常见的板卡。如果希望在一个板卡上使用 RT-Thread 内核，除了需要有相应的芯片架构的移植，还需要有针对板卡的移植，也就是实现一个基本的 BSP。主要任务是建立让操作系统运行的基本环境，需要完成的主要工作是：

- 1) 初始化 CPU 内部寄存器，设定 RAM 工作时序。
- 2) 实现时钟驱动及中断控制器驱动，完善中断管理。
- 3) 实现串口和 GPIO 驱动。
- 4) 初始化动态内存堆，实现动态堆内存管理。

第 10 章

Env 用户手册

Env 是 RT-Thread 推出的开发辅助工具，针对基于 RT-Thread 操作系统的项目工程，提供编译构建环境、图形化系统配置及软件包管理功能。

其内置的 `menuconfig` 提供了简单易用的配置剪裁工具，可对内核、组件和软件包进行自由裁剪，使系统以搭积木的方式进行构建。

10.1 主要特性

- `menuconfig` 图形化配置界面，交互性好，操作逻辑强；
- 丰富的文字帮助说明，配置无需查阅文档；
- 使用灵活，自动处理依赖，功能开关彻底；
- 自动生成 `rtconfig.h`，无需手动修改；
- 使用 `scons` 工具生成工程，提供编译环境，操作简单；
- 提供多种软件包，模块化软件包耦合关联少，可维护性好；
- 软件包可在线下载，软件包持续集成，包可靠性高；

10.2 准备工作

Env 工具包含了 RT-Thread 源代码开发编译环境和软件包管理系统。

- 从 RT-Thread 官网下载 Env 工具。
- 在电脑上装好 git，软件包管理功能需要 git 的支持。git 的下载地址为<https://git-scm.com/downloads>，根据向导正确安装 git，并将 git 添加到系统环境变量。
- 注意在工作环境中，所有的路径都不可以有中文字符或者空格。

10.3 Env 的使用方法

10.3.1 打开 Env 控制台

RT-Thread 软件包环境主要以命令行控制台为主，同时以字符型界面来进行辅助，使得尽量减少修改配置文件的方式即可搭建好 RT-Thread 开发环境的方式。打开 Env 控制台有两种方式：

10.3.1.1 方法一：点击 Env 目录下可执行文件

进入 Env 目录，可以运行本目录下的 `env.exe`，如果打开失败可以尝试使用 `env.bat`。

10.3.1.2 方法二：在文件夹中通过右键菜单打开 Env 控制台

Env 目录下有一张 `Add_Env_To_Right-click_Menu.png`(添加 Env 至右键菜单.png) 的图片，如下：

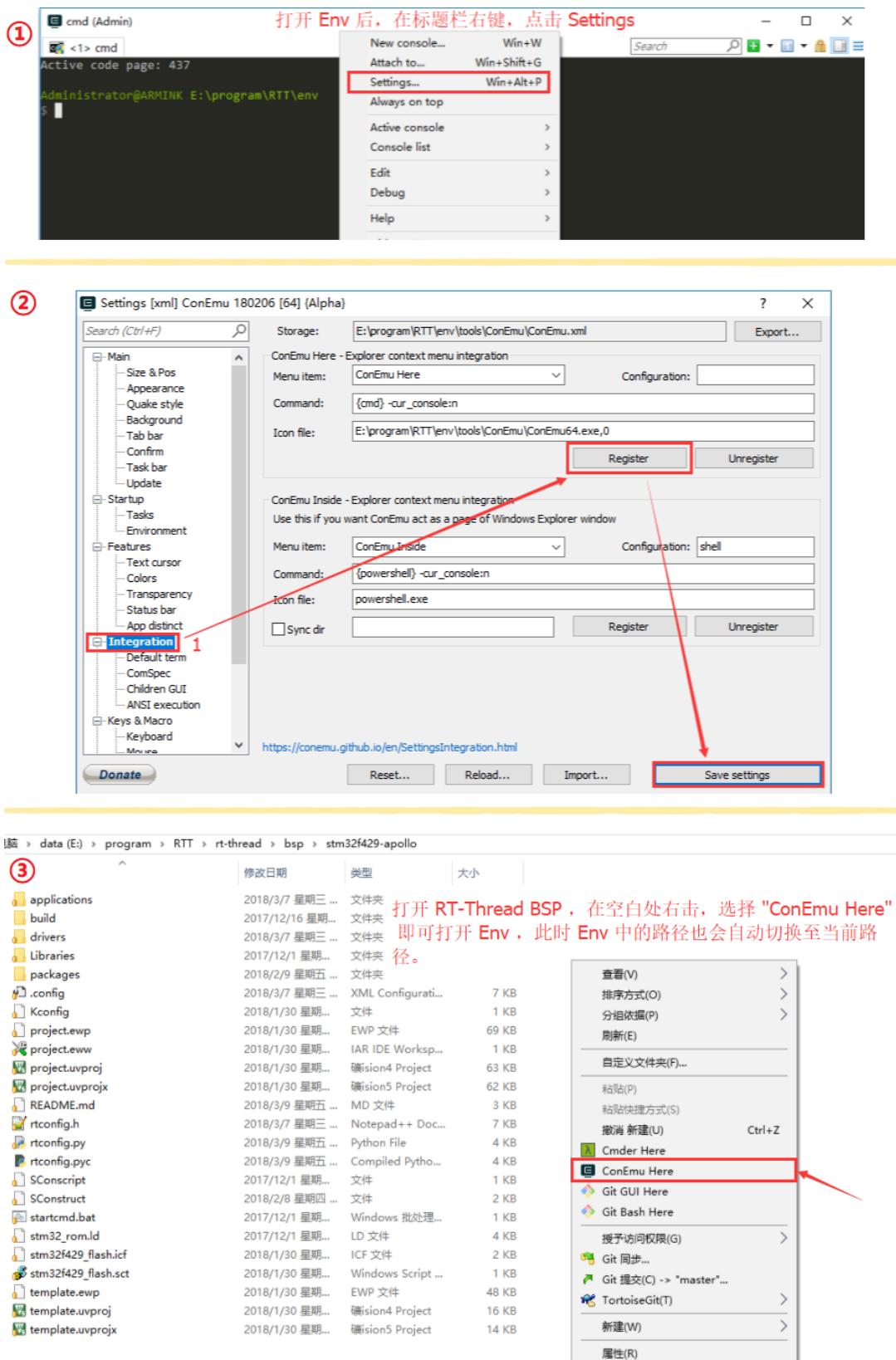


图 10.1: 添加 Env 控制台到右键菜单

根据图片上的步骤操作，就可以在任意文件夹下通过右键菜单来启动 Env 控制台。效果如下：

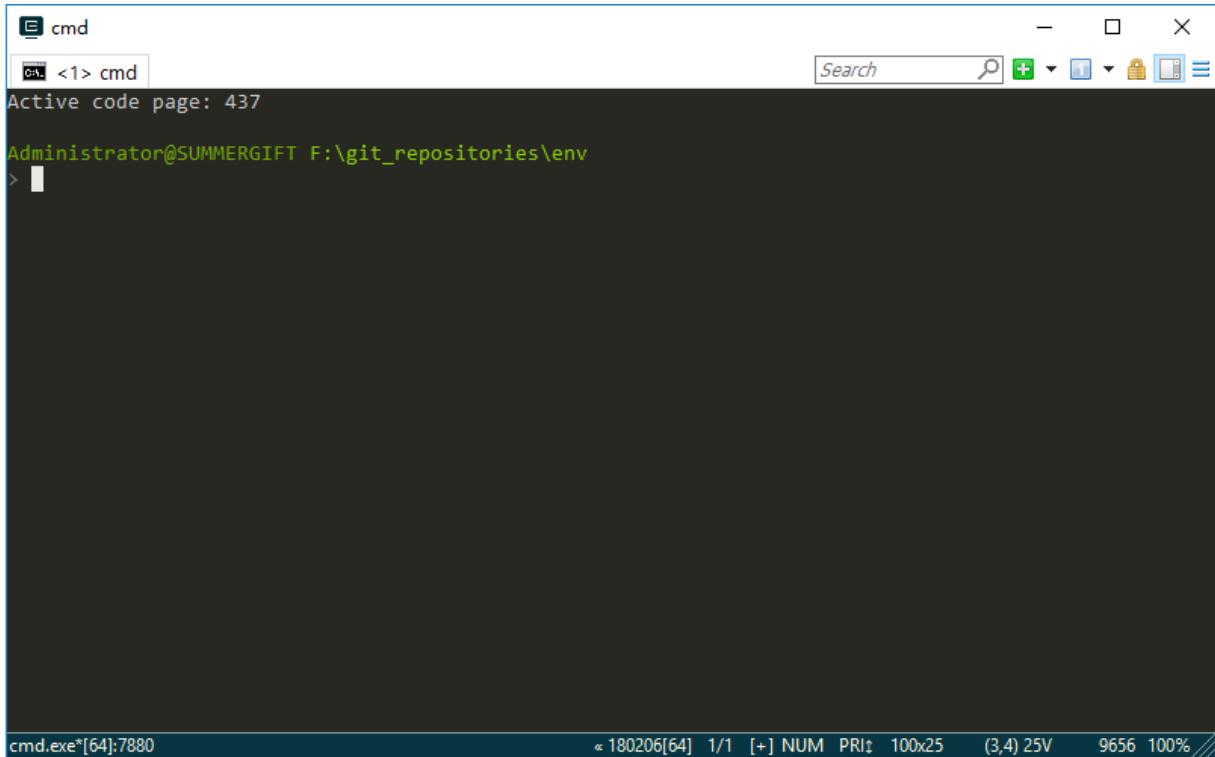


图 10.2: 通过右键菜单来启动 Env 控制台

!!! note “注意事项” 因为需要设置 Env 进程的环境变量，第一次启动可能会出现杀毒软件误报的情况，如果遇到了杀毒软件误报，允许 Env 相关程序运行，然后将相关程序添加至白名单即可。

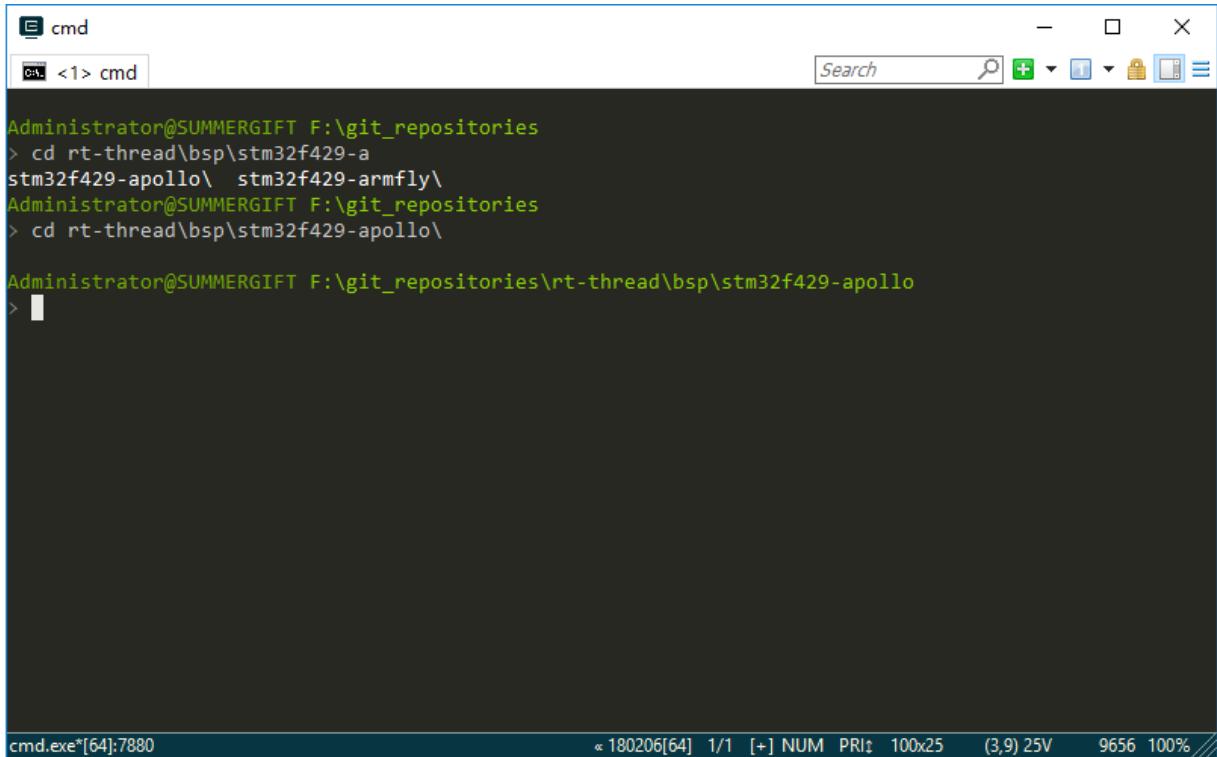
10.3.2 编译 BSP

scons 是 RT-Thread 使用的编译构建工具，可以使用 scons 相关命令来编译 RT-Thread。

10.3.2.1 第一步：切换到 BSP 根目录

- 打开控制台后，可以在命令行模式下使用 cd 命令切换到你想要配置的 BSP 根目录中。

例如工程目录为: `rt-thread\bsp\stm32f429-apollo` :



```
Administrator@SUMMERGIFT F:\git_repositories
> cd rt-thread\bsp\stm32f429-a
stm32f429-apollo\  stm32f429-armfly\
Administrator@SUMMERGIFT F:\git_repositories
> cd rt-thread\bsp\stm32f429-apollo\

Administrator@SUMMERGIFT F:\git_repositories\rt-thread\bsp\stm32f429-apollo
> 
```

图 10.3: *stm32f429-apollo* 工程目录

10.3.2.2 第二步：bsp 的编译

- Env 中携带了 `Python` & `scons` 环境，只需在 `rt-thread\bsp\stm32f429-apollo` 目录中运行 `scons` 命令即可使用默认的 `ARM_GCC` 工具链编译 `bsp`。

```

Administrator@SUMMERGIFT F:\git_repositories
> cd rt-thread\bsp\stm32f429-a
stm32f429-apollo\ stm32f429-armfly\
Administrator@SUMMERGIFT F:\git_repositories
> cd rt-thread\bsp\stm32f429-apollo\

Administrator@SUMMERGIFT F:\git_repositories\rt-thread\bsp\stm32f429-apollo
> scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build

```

python.exe[64]:11164 « 180206[64] 1/1 [+] NUM PRI↑ 100x25 (1,13) 25V 9656 100% //

图 10.4: scons 命令编译工程

编译成功:

```

CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_sai_ex.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_sd.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_sdram.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_smartcard.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_spdifrx.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_spi.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_sram.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_tim.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_tim_ex.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_uart.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_usart.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_hal_wwdg.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_ll_fmc.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_ll_fsmc.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_ll_sdmmc.o
CC build\Libraries\STM32F4xx_HAL_Driver\Src\stm32f4xx_ll_usb.o
LINK rtthread-stm32f42x.axf
arm-none-eabi-objcopy -O binary rtthread-stm32f42x.axf rtthread.bin
arm-none-eabi-size rtthread-stm32f42x.axf
    text      data      bss      dec      hex filename
 305440     2652    63092   371184   5a9f0  rtthread-stm32f42x.axf
scons: done building targets.

Administrator@SUMMERGIFT F:\git_repositories\rt-thread\bsp\stm32f429-apollo
> cmd.exe[64]:7880      « 180206[64] 1/1 [+] NUM PRI↑ 100x25 (3,426) 25V 9656 100% //

```

图 10.5: 编译工程成功

如果使用 mdk/iar 来进行项目开发，可以直接使用 BSP 中的工程文件或者使用以下命令中的其中一

种，重新生成工程，再进行编译下载。

```
scons --target=iar
scons --target=mdk4
scons --target=mdk5
```

更多 scons 教程，请参考《Scons 构建工具》

10.3.3 BSP 配置：menuconfig

menuconfig 是一种图形化配置工具，RT-Thread 使用其对整个系统进行配置、裁剪。

10.3.3.1 快捷键介绍

进入 BSP 根目录，输入 menuconfig 命令后即可打开其界面。menuconfig 常用快捷键如图所示：

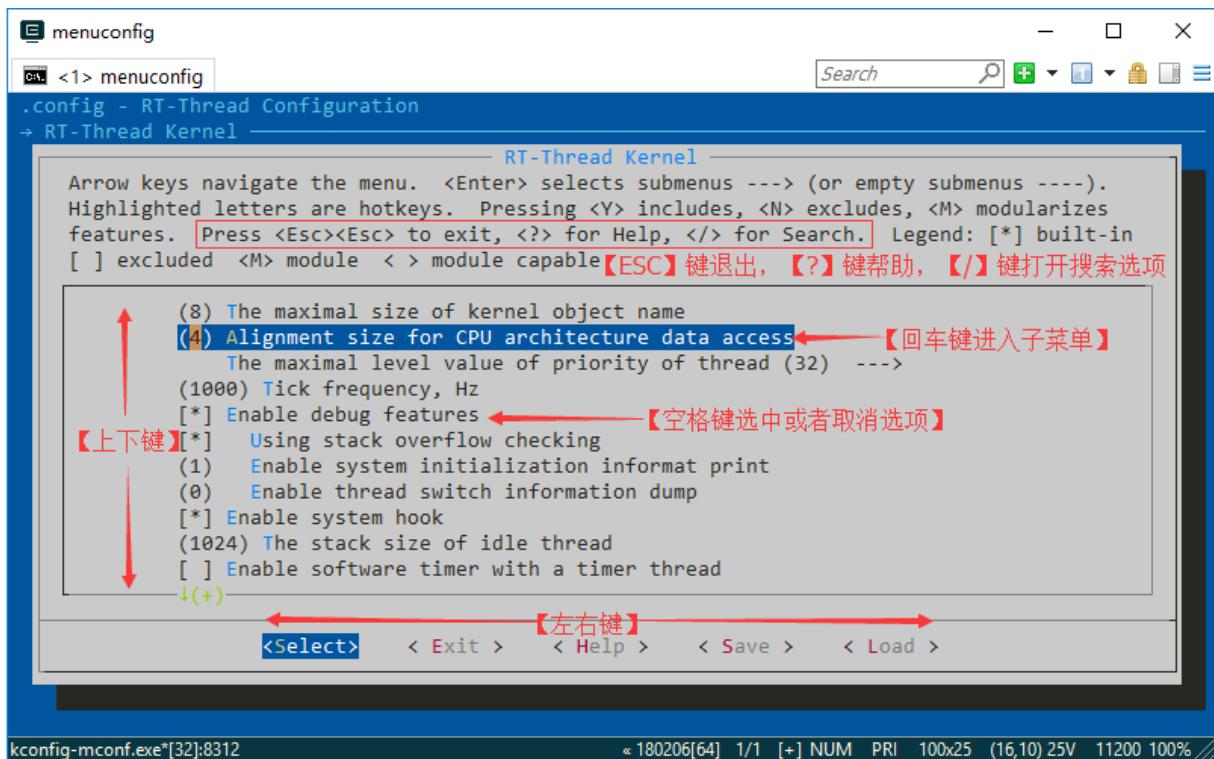


图 10.6: menuconfig 常用快捷键

10.3.3.2 修改配置

menuconfig 有多种类型的配置项，修改方法也有所不同，常见类型如下：

- 开/关型：使用空格键来选中或者关闭
- 数值、字符串型：按下调用键后会出现对话框，在对话框中对配置项进行修改

10.3.3.3 保存配置

选择好配置项之后按 ESC 键退出，选择保存修改即可自动生成 `rtconfig.h` 文件。此时再次使用 `scons` 命令就会根据新的 `rtconfig.h` 文件重新编译工程了。

10.3.4 软件包管理: package

RT-Thread 提供一个软件包管理平台，这里存放了官方提供或开发者提供的软件包。该平台为开发者提供了众多可重用软件包的选择，这也是 RT-Thread 生态的重要组成部分。

[点击这里](#) 可以查看到 RT-Thread 官方的提供的软件包，绝大多数软件包都有详细的说明文档及使用示例。

!!! tip “提示” 截止到 2018-03-13，当前软件包数量达到 40+

package 工具作为 Env 的组成部分，为开发者提供了软件包的下载、更新、删除等管理功能。

Env 命令行输入 `pkgs` 可以看到命令简介：

```
> pkgs
usage: env.py package [-h] [--update] [--list] [--wizard] [--upgrade]
                      [--printenv]

optional arguments:
  -h, --help    show this help message and exit
  --update     update packages, install or remove the packages as you set in
               menuconfig
  --list       list target packages
  --wizard     create a package with wizard
  --upgrade   update local packages list from git repo
  --printenv  print environmental variables to check
```

10.3.4.1 下载、更新、删除软件包

在下载、更新软件包前，需要先在 `menuconfig` 中 [开启](#)你想要操作的软件包

这些软件包位于 `RT-Thread online packages` 菜单下，进入该菜单后，则可以看如下软件包分类：

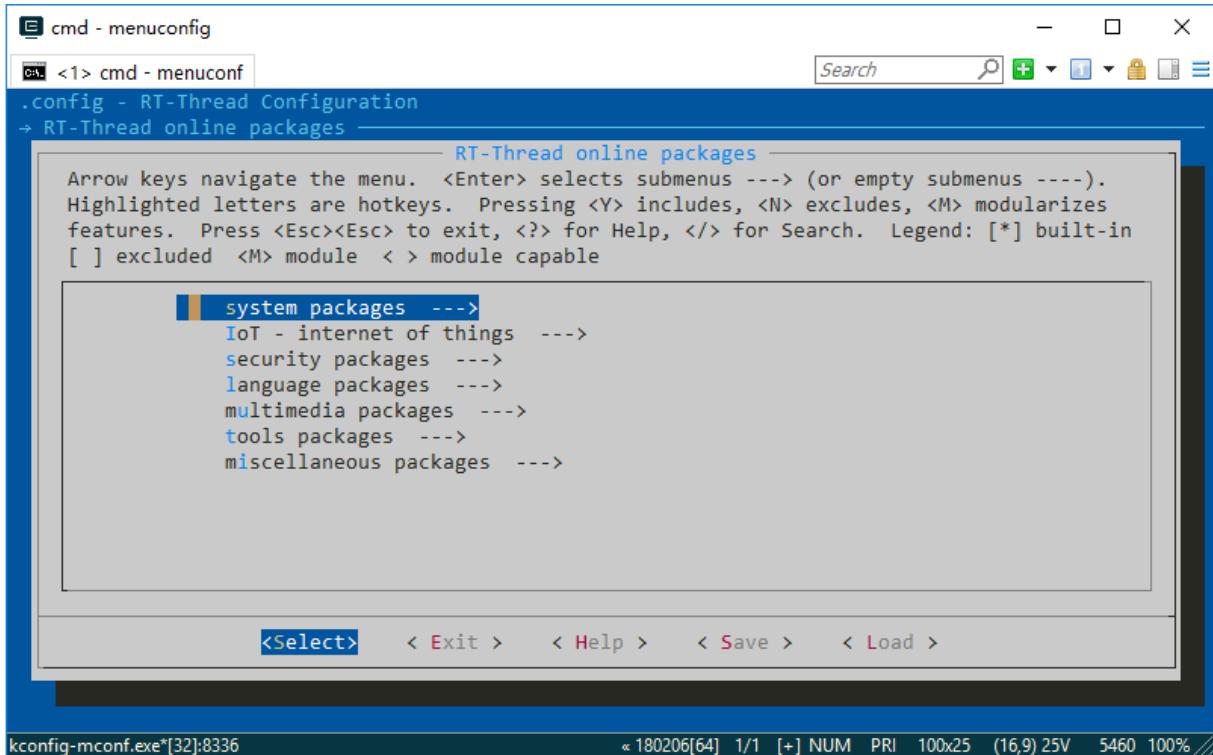


图 10.7: 软件包分类

找到你需要的软件包然后选中开启，保存并退出 `menuconfig`。此时软件包已被标记选中，但是还没有下载到本地，所以还无法使用。

- **下载:** 如果软件包在本地已被选中，但是未下载，此时输入：`pkgs --update`，该软件包自动下载；
- **更新:** 如果选中的软件包在服务器端有更新，并且版本号选择的是 `latest`。此时输入：`pkgs --update`，该软件包将会在本地进行更新；
- **删除:** 某个软件包如果无需使用，需要先在 `menuconfig` 中取消其的选中状态，然后再执行：`pkgs --update`。此时本地已下载但未被选中的软件包将会被删除。

10.3.4.2 升级本地软件包信息

随着 package 系统的不断壮大，会有越来越多的软件包加入进来，所以本地看到 `menuconfig` 中的软件包列表可能会与服务器 **不同步**。使用 `pkgs --upgrade` 命令即可解决该问题，这个命令不仅会对本地的包信息进行更新同步，还会对 Env 的功能脚本进行升级，建议定期使用。

10.3.5 Env 工具配置

- 新版本的 Env 工具中加入了自动更新软件包和自动生成 mdk/iar 工程的选项，默认是不开启的。可以使用 `menuconfig -s/--setting` 命令来进行配置。
- 使用 `menuconfig -s` 命令进入 Env 配置界面

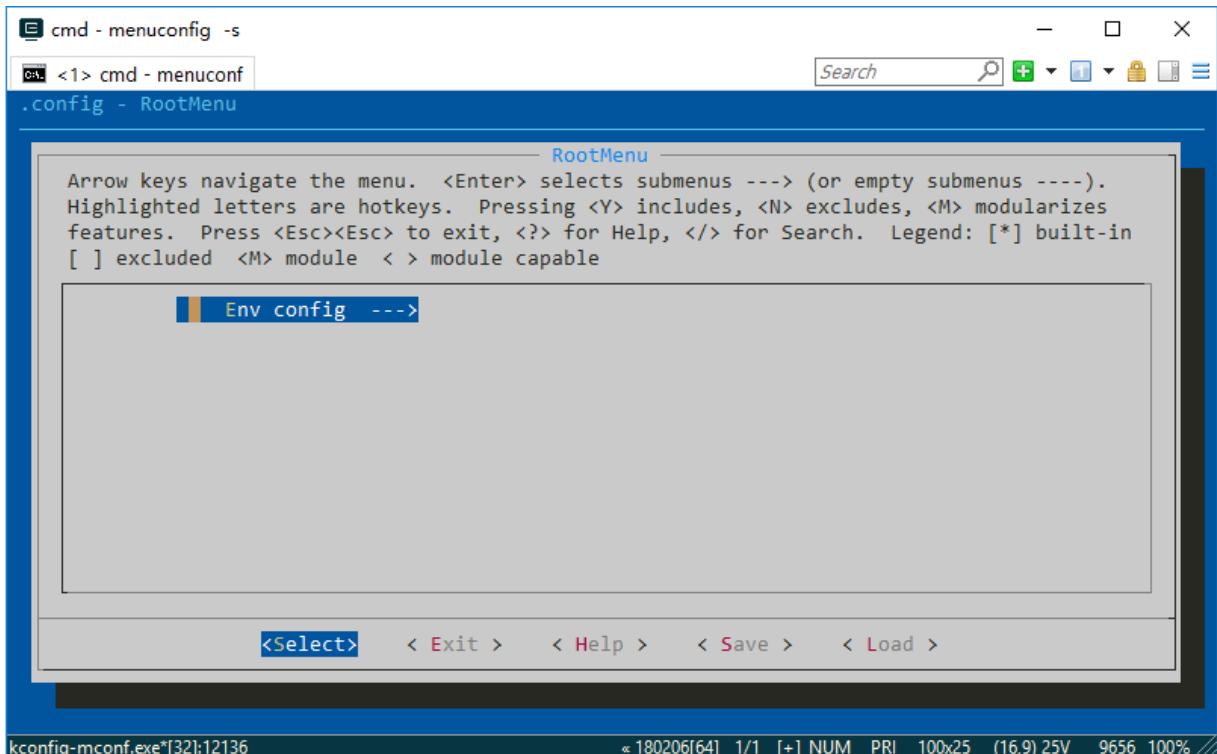


图 10.8: Env 配置界面

按下回车进入配置菜单，里面共有 3 个配置选项

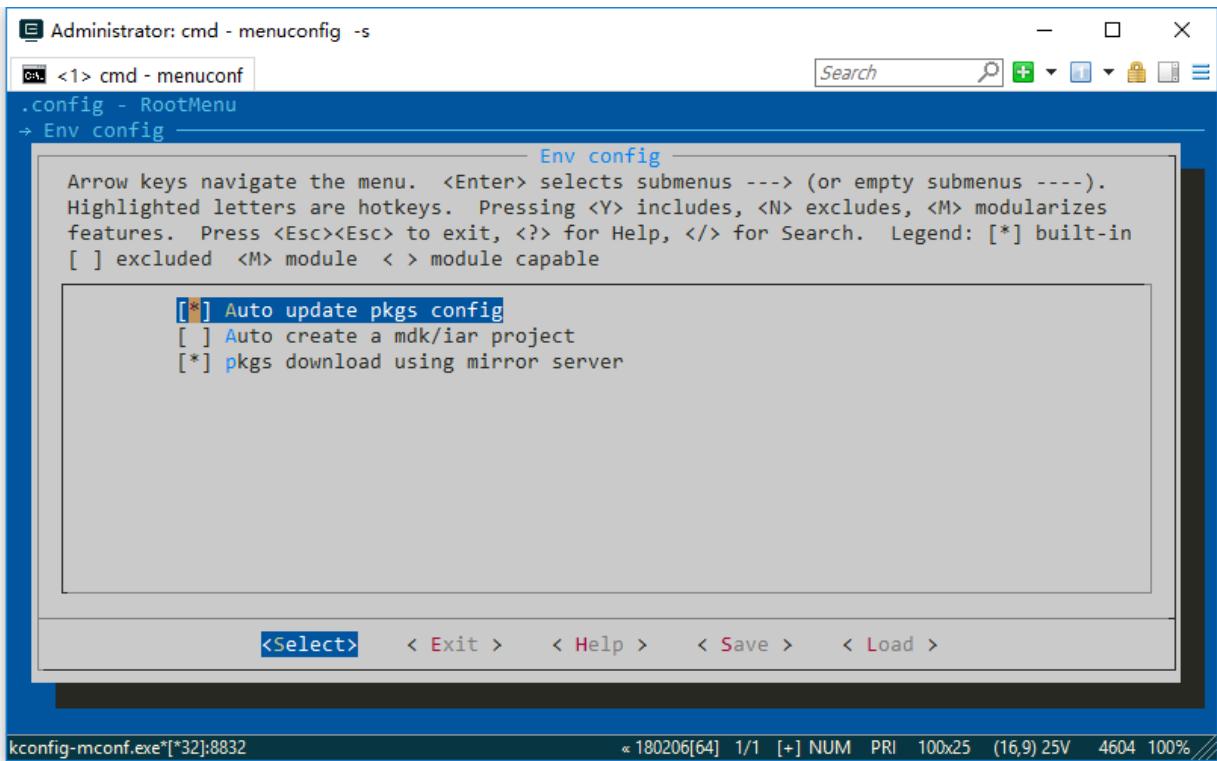


图 10.9: 配置选项

3 个选项分别为：

- **软件包自动更新功能：**在退出 menuconfig 功能后，会自动使用`pkgs --update`命令来下载并安装软件包，同时删除旧的软件包。本功能在下载在线软件包时使用。
- **自动创建 MDK 或 IAR 工程功能：**当修改 menuconfig 配置后，必须输入`scons --target=xxx`来重新生成工程。开启此功能，就会在退出 menuconfig 时，自动重新生成工程，无需再手动输入`scons`命令来重新生成工程。
- **使用镜像服务器下载软件包：**由于大部分软件包目前均存放在 GitHub 上，所以在国内的特殊环境下，下载体验非常差。开启此功能，可以通过国内镜像服务器下载软件包，大幅提高软件包的下载速度和稳定性，减少更新软件包和 submodule 时的等待时间，提升下载体验。

10.4 在项目中使用 Env

10.4.1 使用 Env 的要求

- menuconfig 是 RT-Thread 3.0 以上版本的特性，推荐将 RT-Thread 更新到 3.0 以上版本。
- 目前 RT-Thread 还没有对所有的 BSP 做 menuconfig 的支持，也就是说有些 BSP 暂时还不能使用 menuconfig 来进行配置，但常用的 BSP 都已经支持。

10.4.2 menuconfig 中选项的修改方法

如果想在 menuconfig 的配置项中添加宏定义，则可以修改 BSP 下的 Kconfig 文件，修改方法可以在网络中搜索[Kconfig语法](#)关键字获得详细的说明文档，也可以参考 RT-Thread 中的 Kconfig 文件或者已经支持过 menuconfig 的 BSP 中的 Kconfig 文件。

10.4.3 新的项目添加 menuconfig 功能

这里的个项目指的是，还未生成.config 和 rtconfig.h 的全新开发的项目。因为这两个文件，只有在 menuconfig 第一次保存时才会创建。具体流程如下：

1. 将已经支持 menuconfig 功能的 BSP 里面的 kconfig 文件拷贝到新的项目根目录中。
2. 注意修改 Kconfig 中的 RTT_ROOT 值为 RT-Thread 所在目录，否则可能提示找不到 RTT_ROOT。
3. 使用 menuconfig 命令开始配置即可。

10.4.4 旧项目添加 menuconfig 功能

这里的旧项目指的是已经经过一段时间的开发，而且项目中存在已经修改过的 rtconfig.h 文件，但是没有使用过 menuconfig 来配置的项目。具体流程如下：

1. 首先备份旧项目内的 rtconfig.h 文件。
2. 使用`scons --genconfig`命令根据已有的 rtconfig.h 生成.config 文件，这里生成的.config 文件保存了旧项目中 rtconfig.h 文件对项目的配置参数。
3. 将已经支持 menuconfig 功能的 BSP 里面的 kconfig 文件拷贝到要修改项目的根目录中。
4. 注意修改 Kconfig 中的 RTT_ROOT 值为 RT-Thread 所在目录，否则可能提示找不到 RTT_ROOT。

5. 使用 `menuconfig` 命令来配置我们要修改的旧项目。`menuconfig` 会读取第二步生成的.config 文件，并根据旧项目的配置参数生成新的.config 文件和 `rtconfig.h` 文件。
6. 对比检查新旧两份 `rtconfig.h` 文件，如果有不一致的地方，可以使用 `menuconfig` 命令对配置项进行调整。

10.4.5 用户软件包管理功能

实际开发项目时，开发者可能想要将已下载的软件包加入 git 管理，或者想自己管理该软件包。不希望 Env 工具再拉取该软件包的最新版本，此时可以使用用户软件包管理功能。

如果用户手动将 `EasyFlash-v4.1.0` 文件夹的后缀，也就是软件包的版本号删除，修改为 `EasyFlash`，此时再次使用 `pkgs --update` 命令将不会再拉取 `EasyFlash-v4.0.0` 软件包。Env 工具此时认为 `EasyFlash` 软件包由用户管理，此时使用 `pkgs --force-update` 命令才可以重新拉取附带 version 的新版本软件包。

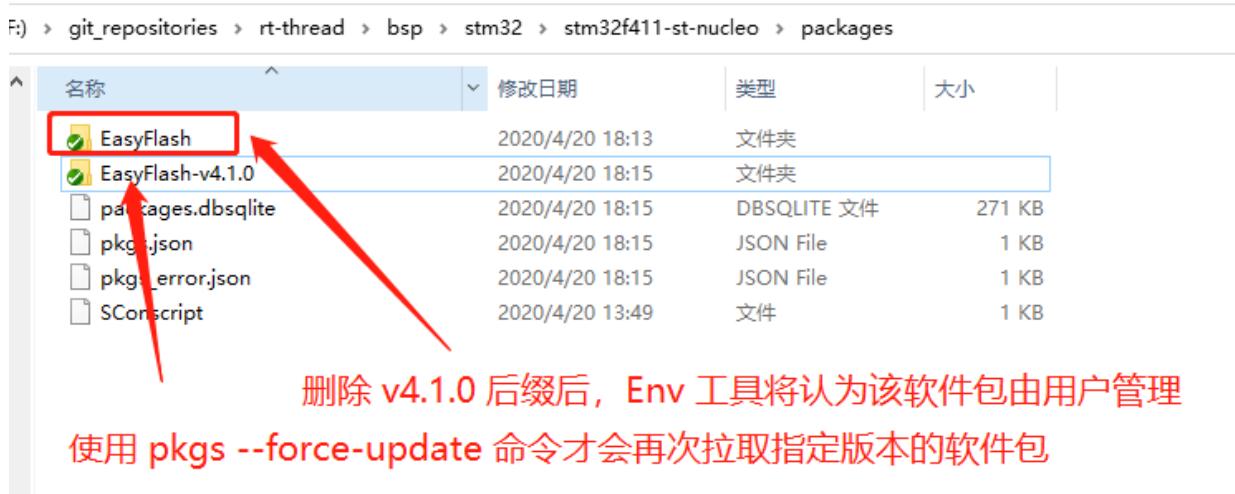


图 10.10: user_manage_package

10.5 使用 pip 扩展更多功能

在 Env 环境下暂时不能直接使用 Python 提供的 pip 工具来安装更多模块。如果需要在 Env 环境下使用 pip 功能，可以按照如下方法重新安装 pip 工具：

1. 从地址 <https://bootstrap.pypa.io/get-pip.py> 下载 `get-pip.py` 文件，存放在磁盘中。
2. 在 Env 环境下执行 `python get-pip.py` 命令来重新安装 pip 工具。
3. pip 工具重新安装成功后，可以使用 `pip install module-name` 命令来安装所需模块。

10.6 Env 工具使用注意事项

!!! note “注意事项” - 第一次使用 Env 推荐去官网下载最新版本的 Env 工具，新版本的 Env 会有更好的兼容性，也支持自动更新的命令。- 可以使用 Env 内置命令 `pkgs -upgrade` 来更新软件包列表和 Env 的

功能代码，这样可以最大程度避免遇到已经修复的问题。- Env 所在路径不要有中文或者空格存在。- BSP 工程所在的路径不要有中文或者空格存在。

10.7 常见问题

10.7.1 Q: Env 工具出现乱码怎么办？

A: 首先检查是否有中文路径。检查 chcp 命令是否加入了系统环境变量，尝试使用 chcp 437 命令将字符格式改为英文。如果提示没有 chcp 命令，则考虑是没有加入到环境变量中。chcp 命令所在的目录可能在 system32 目录，添加到环境变量即可。[Env 工具乱码问题传送门](#)。

10.7.2 Q: 提示找不到 git 命令？

‘git’ is not recognized as an internal or external command, operable program or batch file.

A: 没安装 git，需要安装 git 并加入环境变量。

10.7.3 Q: 提示找不到 CMD 命令？

A: 计算机右键->> 属性—>> 高级系统设置—>> 环境变量，[C:\Windows\System32](#); 加入系统环境变量即可

10.7.4 Q: 运行 python 的时候提示 no module named site 怎么办？

A: 计算机右键->> 属性—>> 高级系统设置—>> 环境变量，在管理员的用户变量中，新建变量名为 PYTHONHOME，变量值为：[F:\git_repositories\env\tools\Python27](#)（是 Env 里面 Python 的安装路径），注意后面不要加“；”，否则会无效。如果添加 PYTHONHOME 没好，再用同样的方法添加 PYTHONPATH。就可以解决这个问题了。

有一篇博文详细的描述了这个问题：[传送门在这里](#)，如果想了解原理可以看一看。

10.7.5 Q: 在 Env 下能生成哪些类型的工程？

A:

1. 目前在 Env 下可以使用 scons 工具生成 mdk/iar 的工程，还没有支持 eclipse 工程的自动生成。
2. 一般在使用 Env 的开发，使用 gcc 的工具链，那么只需要一个 source insight 或者 vs code 之类的编辑器来看代码，使用 scons 编译即可。

10.7.6 Q: 自己制作的 BSP 如何能支持 menuconfig？

A: 可以查阅本章 在项目中使用 Env 章节。

10.7.7 Q: pkgs -upgrade 命令和 pkgs -update 命令有什么区别?

A:

1. pkgs -upgrade 命令是用来升级 Env 功能脚本本身和软件包列表的。没有最新的包列表就不能选择最近更新的软件包。
2. pkgs -update 命令是用来更新软件包本身的，比如说你在 menuconfig 中选中了 json 和 mqtt 的软件包，但是退出 menuconfig 时并没有下载这些软件包。你需要使用 pkgs -update 命令，这时候 Env 就会下载你选中的软件包并且加入到你的工程中去。
3. 新版本的 Env 支持 menuconfig -s/-setting 命令，如果你不想每次更换软件包后使用 pkgs -update 命令，在使用 menuconfig -s/-setting 命令后配置 Env 选择每次使用 menuconfig 后自动更新软件包即可。

10.7.8 Q: VC98 文件夹问题

详细描述：出现错误 MissingConfiguration: registry dir D:\Program Files (x86)\Microsoft Visual Studio\VC98 not found on the filesystem

```
E:\Learn_More\STM32\RT_Thread\Code\rt-thread\bsp\stm32f429-apollo
> scons
scons: Reading SConscript files ...
MissingConfiguration: registry dir D:\Program Files (x86)\Microsoft Visual Studio\VC98 not found on the filesystem:
  File "E:\Learn_More\STM32\RT_Thread\Code\rt-thread\bsp\stm32f429-apollo\SConstruct", line 32:
    Export('RTT_ROOT')
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Script\SConscript.py", line 612:
    env = self.factory()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Script\SConscript.py", line 592:
    default_env = SCons.DefaultEnvironment()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Defaults.py", line 88:
    _default_env = SCons.Environment.Environment(*args, **kw)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Environment.py", line 1006:
    apply_tools(self, tools, toolpath)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Environment.py", line 107:
    env.Tool(tool)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Environment.py", line 1814:
    tool(self)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\__init__.py", line 183:
    self.generate(env, *args, **kw)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\default.py", line 40:
    for t in SCons.Tool.tool_list(env['PLATFORM'], env):
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\__init__.py", line 805:
    linker = FindTool(linkers, env) or linkers[0]
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\__init__.py", line 692:
    if t.exists(env):
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\linkloc.py", line 103:
    if msvs_exists():
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 445:
    return (len(get_installed_visual_studios()) > 0)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 393:
    if vs.get_executable():
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 134:
    executable = self.find_executable()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 108:
    vs_dir = self.get_vs_dir()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 143:
    vs_dir = self.find_vs_dir()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 102:
    vs_dir=self.find_vs_dir_by_reg()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 84:
    return self.find_vs_dir_by_vc()
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vs.py", line 71:
    dir = SCons.Tool.MSCommon_vc.find_vc_pdir(self_vc_version)
  File "E:\Learn_More\STM32\RT_Thread\env\tools\Python27\Scripts..\Lib\site-packages\scons-2.3.6\SCons\Tool\MSCommon\vc.py", line 240:
    raise MissingConfiguration("registry dir %s not found on the filesystem" % comps)
```

图 10.11: VC98 问题

A: 在划线的目录新建一个 VC98 的空文件夹，就可以使用 scons 了。

10.7.9 Q: 使用 menuconfig 命令提示 “can't find file Kconfig”。

A: 当前工作的 BSP 目录下缺少 Kconfig 文件，参考本文《新的项目添加 menuconfig 功能》和《旧项目添加 menuconfig 功能》。

10.7.10 Q: IOError: [Errno 2] No such file or directory: ‘nul’

A: 这是由于 windows 系统没有开启 Null Service 服务的缘故，常见于在 win10 的早期版本中（如版本号 1703），该问题有两种解决方法，第一种是开启 windows 更新将 windows 更新到最新版本，因为在后续的补丁中 windows 默认开启了该服务，第二种是参考该 [link](#) 手动开启 Null Service 服务。

10.8 常用资料链接

- 论坛持续更新的 Env 常见问题问答帖

第 11 章

I/O 设备模型

绝大部分的嵌入式系统都包括一些 I/O (Input/Output, 输入 / 输出) 设备，例如仪器上的数据显示屏、工业设备上的串口通信、数据采集设备上用于保存数据的 Flash 或 SD 卡，以及网络设备的以太网接口等，都是嵌入式系统中容易找到的 I/O 设备例子。

本章主要介绍 RT-Thread 如何对不同的 I/O 设备进行管理，读完本章，我们会了解 RT-Thread 的 I/O 设备模型，并熟悉 I/O 设备管理接口的不同功能。

11.1 I/O 设备介绍

11.1.1 I/O 设备模型框架

RT-Thread 提供了一套简单的 I/O 设备模型框架，如下图所示，它位于硬件和应用程序之间，共分成三层，从上到下分别是 I/O 设备管理层、设备驱动框架层、设备驱动层。

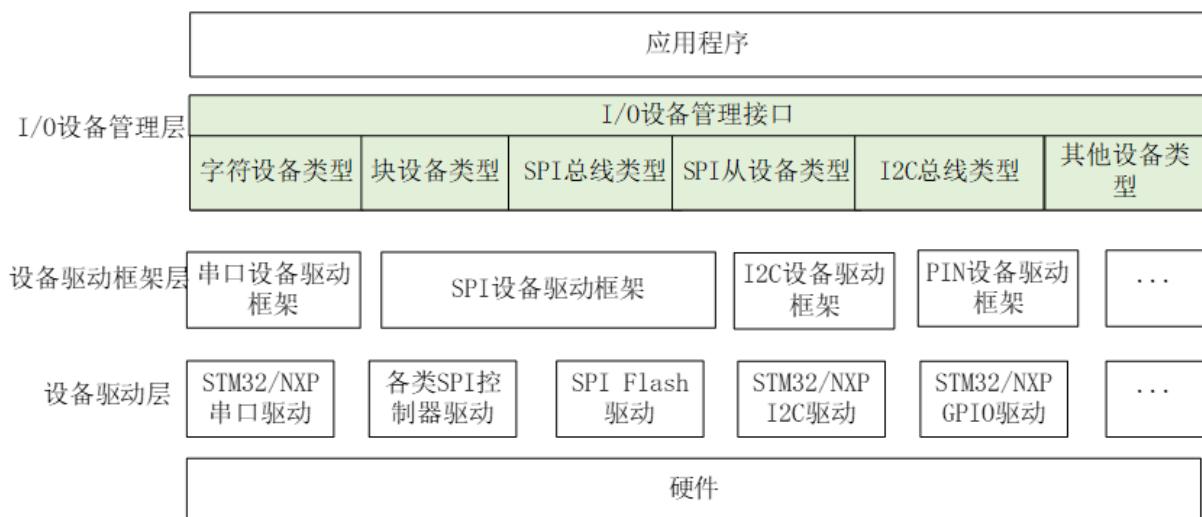


图 11.1: I/O 设备模型框架

应用程序通过 I/O 设备管理接口获得正确的设备驱动，然后通过这个设备驱动与底层 I/O 硬件设备进行数据（或控制）交互。

I/O 设备管理层实现了对设备驱动程序的封装。应用程序通过 I/O 设备层提供的标准接口访问底层设备，设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在，双方只需关注各自的功能实现，从而降低了代码的耦合性、复杂性，提高了系统的可靠性。

设备驱动框架层是对同类硬件设备驱动的抽象，将不同厂家的同类硬件设备驱动中相同的部分抽取出 来，将不同部分留出接口，由驱动程序实现。

设备驱动层是一组驱使硬件设备工作的程序，实现访问硬件设备的功能。它负责创建和注册 I/O 设备，对于操作逻辑简单的设备，可以不经过设备驱动框架层，直接将设备注册到 I/O 设备管理器中，使用序列图如下图所示，主要有以下 2 点：

- 设备驱动根据设备模型定义，创建出具备硬件访问能力的设备实例，将该设备通过 `rt_device_register()` 接口注册到 I/O 设备管理器中。
- 应用程序通过 `rt_device_find()` 接口查找到设备，然后使用 I/O 设备管理接口来访问硬件。

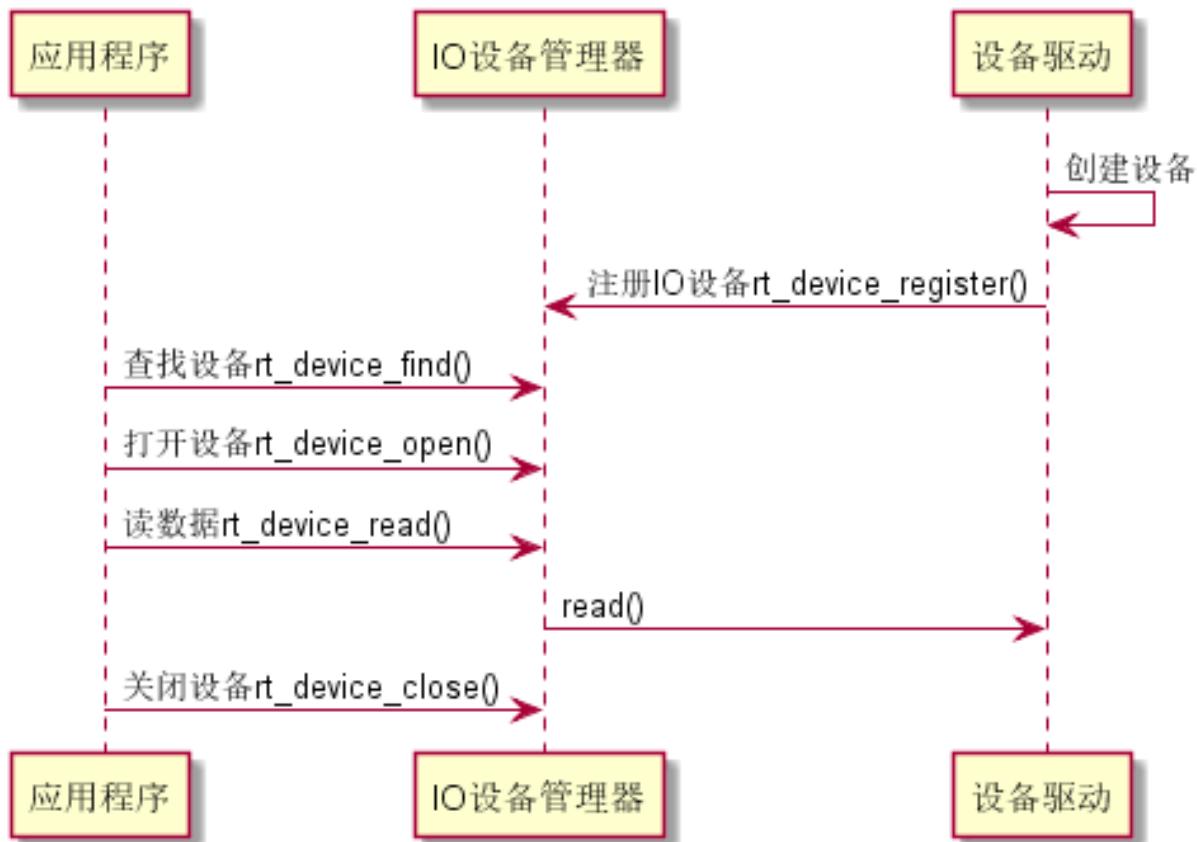


图 11.2: 简单 I/O 设备使用序列图

对于另一些设备，如看门狗等，则会将创建的设备实例先注册到对应的设备驱动框架中，再由设备驱动框架向 I/O 设备管理器进行注册，主要有以下几点：

- 看门狗设备驱动程序根据看门狗设备模型定义，创建出具备硬件访问能力的看门狗设备实例，并将该看门狗设备通过 `rt_hw_watchdog_register()` 接口注册到看门狗设备驱动框架中。
- 看门狗设备驱动框架通过 `rt_device_register()` 接口将看门狗设备注册到 I/O 设备管理器中。

- 应用程序通过 I/O 设备管理接口来访问看门狗设备硬件。

看门狗设备使用序列图:

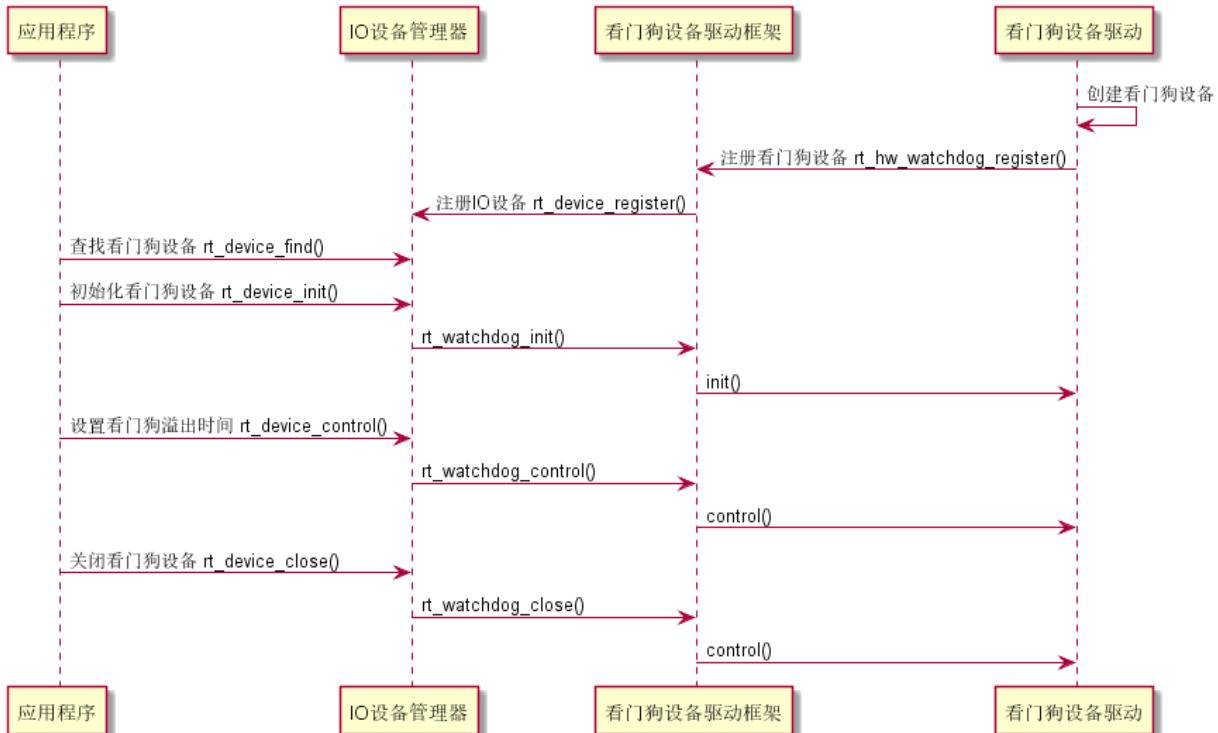


图 11.3: 看门狗设备使用序列图

11.1.2 I/O 设备模型

RT-Thread 的设备模型是建立在内核对象模型基础之上的，设备被认为是一类对象，被纳入对象管理器的范畴。每个设备对象都是由基对象派生而来，每个具体设备都可以继承其父类对象的属性，并派生出其私有属性，下图是设备对象的继承和派生关系示意图。

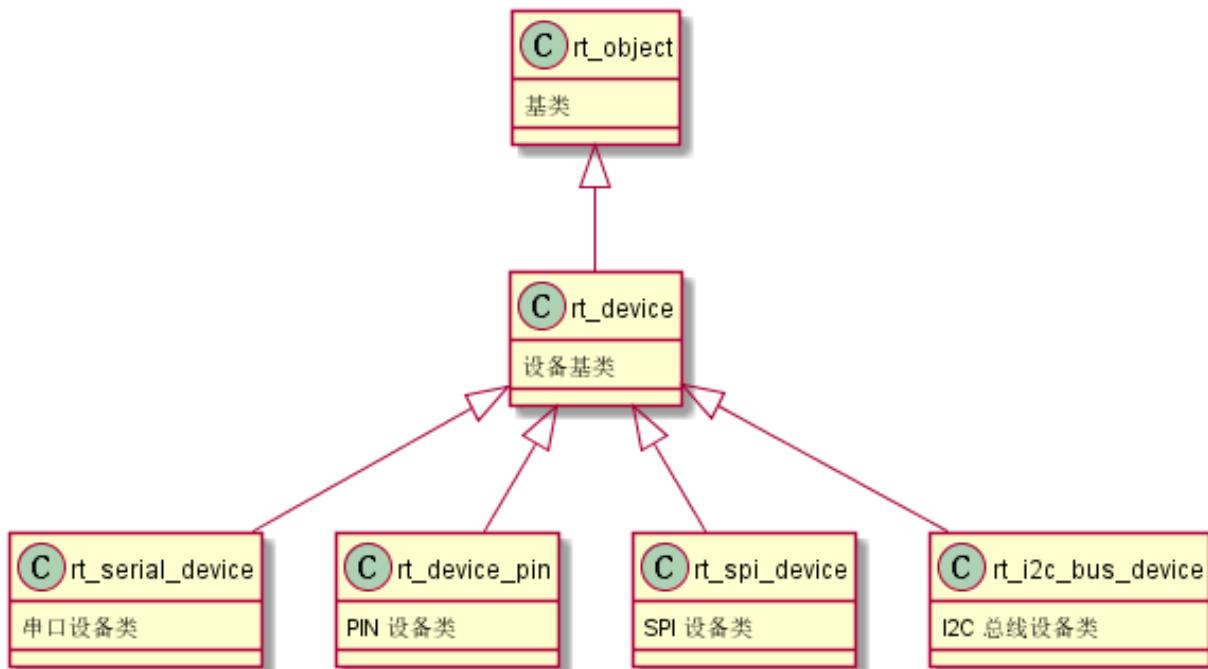


图 11.4: 设备继承关系图

设备对象具体定义如下所示：

```

struct rt_device
{
    struct rt_object          parent;      /* 内核对象基类 */
    enum rt_device_class_type type;        /* 设备类型 */
    rt_uint16_t                flag;        /* 设备参数 */
    rt_uint16_t                open_flag;   /* 设备打开标志 */
    rt_uint8_t                 ref_count;   /* 设备被引用次数 */
    rt_uint8_t                 device_id;   /* 设备 ID, 0 - 255 */

    /* 数据收发回调函数 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void *buffer);

    const struct rt_device_ops *ops;      /* 设备操作方法 */
};

/* 设备的私有数据 */
void *user_data;
};

typedef struct rt_device *rt_device_t;
  
```

11.1.3 I/O 设备类型

RT-Thread 支持多种 I/O 设备类型，主要设备类型如下所示：

RT_Device_Class_Char	/* 字符设备 */
----------------------	------------

RT_Device_Class_Block	/* 块设备 */
RT_Device_Class_NetIf	/* 网络接口设备 */
RT_Device_Class_MTD	/* 内存设备 */
RT_Device_Class_RTC	/* RTC 设备 */
RT_Device_Class_Sound	/* 声音设备 */
RT_Device_Class_Graphic	/* 图形设备 */
RT_Device_Class_I2CBUS	/* I2C 总线设备 */
RT_Device_Class_USBDevice	/* USB device 设备 */
RT_Device_Class_USBHost	/* USB host 设备 */
RT_Device_Class_SPIBUS	/* SPI 总线设备 */
RT_Device_Class_SPIDevice	/* SPI 设备 */
RT_Device_Class_SDIO	/* SDIO 设备 */
RT_Device_Class_Miscellaneous	/* 杂类设备 */

其中字符设备、块设备是常用的设备类型，它们的分类依据是设备数据与系统之间的传输处理方式。字符模式设备允许非结构的数据传输，即通常数据传输采用串行的形式，每次一个字节。字符设备通常是一些简单设备，如串口、按键。

块设备每次传输一个数据块，例如每次传输 512 个字节数据。这个数据块是硬件强制性的，数据块可能使用某类数据接口或某些强制性的传输协议，否则就可能发生错误。因此，有时块设备驱动程序对读或写操作必须执行附加的工作，如下图所示：

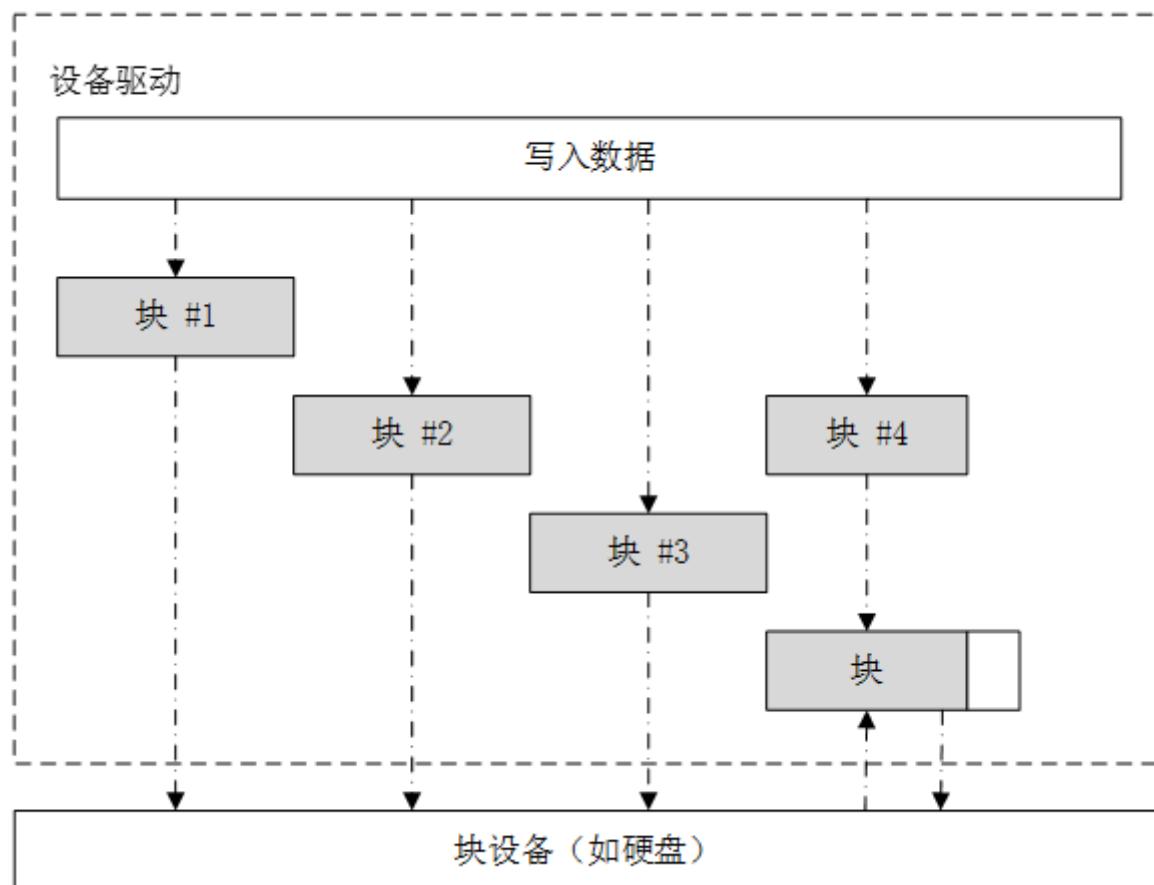


图 11.5: 块设备

当系统服务于一个具有大量数据的写操作时，设备驱动程序必须首先将数据划分为多个包，每个包采用设备指定的数据尺寸。而在实际过程中，最后一部分数据尺寸有可能小于正常的设备块尺寸。如上图中每个块使用单独的写请求写入到设备中，头 3 个直接进行写操作。但最后一个数据块尺寸小于设备块尺寸，设备驱动程序必须使用不同于前 3 个块的方式处理最后的数据块。通常情况下，设备驱动程序需要首先执行相对应的设备块的读操作，然后把写入数据覆盖到读出数据上，然后再把这个“合成”的数据块作为一整个块写回到设备中。例如上图中的块 4，驱动程序需要先把块 4 所对应的设备块读出来，然后将需要写入的数据覆盖至从设备块读出的数据上，使其合并成一个新的块，最后再写回到块设备中。

11.2 创建和注册 I/O 设备

驱动层负责创建设备实例，并注册到 I/O 设备管理器中，可以通过静态申明的方式创建设备实例，也可以用下面的接口进行动态创建：

```
rt_device_t rt_device_create(int type, int attach_size);
```

参数	描述
type	设备类型，可取前面小节列出的设备类型值
attach_size	用户数据大小
返回	—
设备句柄	创建成功
RT_NULL	创建失败，动态内存分配失败

调用该接口时，系统会从动态堆内存中分配一个设备控制块，大小为 struct rt_device 和 attach_size 的和，设备的类型由参数 type 设定。设备被创建后，需要实现它访问硬件的操作方法。

```
struct rt_device_ops
{
    /* common device interface */
    rt_err_t (*init)    (rt_device_t dev);
    rt_err_t (*open)    (rt_device_t dev, rt_uint16_t oflag);
    rt_err_t (*close)   (rt_device_t dev);
    rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t size
    );
    rt_size_t (*write)  (rt_device_t dev, rt_off_t pos, const void *buffer,
    rt_size_t size);
    rt_err_t (*control)(rt_device_t dev, int cmd, void *args);
};
```

各个操作方法的描述如下表所示：

方法名称	方法描述
init	初始化设备。设备初始化完成后，设备控制块的 <code>flag</code> 会被置成已激活状态 (<code>RT_DEVICE_FLAG_ACTIVATED</code>)。如果设备控制块中的 <code>flag</code> 标志已经设置成激活状态，那么再运行初始化接口时会立刻返回，而不会重新进行初始化。
open	打开设备。有些设备并不是系统一启动就已经打开开始运行，或者设备需要进行数据收发，但如果上层应用还未准备好，设备也不应默认已经使能并开始接收数据。所以建议在写底层驱动程序时，在调用 <code>open</code> 接口时才使能设备。
close	关闭设备。在打开设备时，设备控制块会维护一个打开计数，在打开设备时进行 +1 操作，在关闭设备时进行 -1 操作，当计数器变为 0 时，才会进行真正的关闭操作。
read	从设备读取数据。参数 <code>pos</code> 是读取数据的偏移量，但是有些设备并不一定需要指定偏移量，例如串口设备，设备驱动应忽略这个参数。而对于块设备来说， <code>pos</code> 以及 <code>size</code> 都是以块设备的数据块大小为单位的。例如块设备的数据块大小是 512，而参数中 <code>pos = 10, size = 2</code> ，那么驱动应该返回设备中第 10 个块（从第 0 个块做为起始），共计 2 个块的数据。这个接口返回的类型是 <code>rt_size_t</code> ，即读到的字节数或块数目。正常情况下应该会返回参数中 <code>size</code> 的数值，如果返回零请设置对应的 <code>errno</code> 值。
write	向设备写入数据。参数 <code>pos</code> 是写入数据的偏移量。与读操作类似，对于块设备来说， <code>pos</code> 以及 <code>size</code> 都是以块设备的数据块大小为单位的。这个接口返回的类型是 <code>rt_size_t</code> ，即真实写入数据的字节数或块数目。正常情况下应该会返回参数中 <code>size</code> 的数值，如果返回零请设置对应的 <code>errno</code> 值。
control	根据 <code>cmd</code> 命令控制设备。命令往往是由底层各类设备驱动自定义实现。例如参数 <code>RT_DEVICE_CTRL_BLK_GETGEME</code> ，意思是获取块设备的大小信息。

当一个动态创建的设备不再需要使用时可以通过如下函数来销毁：

```
void rt_device_destroy(rt_device_t device);
```

参数	描述
device	设备句柄
返回	无

设备被创建后，需要注册到 I/O 设备管理器中，应用程序才能够访问，注册设备的函数如下所示：

```
rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags);
```

参数	描述
dev	设备句柄
name	设备名称，设备名称的最大长度由 <code>rtconfig.h</code> 中定义的宏 <code>RT_NAME_MAX</code> 指定，多余部分会被自动截掉
flags	设备模式标志

参数	描述
返回	——
RT_EOK	注册成功
-RT_ERROR	注册失败, dev 为空或者 name 已经存在

!!! note “注意事项” 应当避免重复注册已经注册的设备，以及注册相同名字的设备。

flags 参数支持下列参数(可以采用或的方式支持多种参数):

#define RT_DEVICE_FLAG_RDONLY	0x001 /* 只读 */
#define RT_DEVICE_FLAG_WRONLY	0x002 /* 只写 */
#define RT_DEVICE_FLAG_RDWR	0x003 /* 读写 */
#define RT_DEVICE_FLAG_REMOVABLE	0x004 /* 可移除 */
#define RT_DEVICE_FLAG_STANDALONE	0x008 /* 独立 */
#define RT_DEVICE_FLAG_SUSPENDED	0x020 /* 挂起 */
#define RT_DEVICE_FLAG_STREAM	0x040 /* 流模式 */
#define RT_DEVICE_FLAG_INT_RX	0x100 /* 中断接收 */
#define RT_DEVICE_FLAG_DMA_RX	0x200 /* DMA 接收 */
#define RT_DEVICE_FLAG_INT_TX	0x400 /* 中断发送 */
#define RT_DEVICE_FLAG_DMA_TX	0x800 /* DMA 发送 */

设备流模式 RT_DEVICE_FLAG_STREAM 参数用于向串口终端输出字符串：当输出的字符是 “\n” 时，自动在前面补一个 “\r” 做分行。

注册成功的设备可以在 FinSH 命令行使用 `list_device` 命令查看系统中所有的设备信息，包括设备名称、设备类型和设备被打开次数：

```
msh />list_device
device          type      ref count
-----
e0             Network Interface   0
sd0            Block Device     1
rtc             RTC           0
uart1           Character Device 0
uart0           Character Device 2
msh />
```

当设备注销后的，设备将从设备管理器中移除，也就不能再通过设备查找搜索到该设备。注销设备不会释放设备控制块占用的内存。注销设备的函数如下所示：

```
rt_err_t rt_device_unregister(rt_device_t dev);
```

参数	描述
dev	设备句柄
返回	——

参数	描述
RT_EOK	成功

下面代码为看门狗设备的注册示例，调用 `rt_hw_watchdog_register()` 接口后，设备通过 `rt_device_register()` 接口被注册到 I/O 设备管理器中。

```
const static struct rt_device_ops wdt_ops =
{
    rt_watchdog_init,
    rt_watchdog_open,
    rt_watchdog_close,
    RT_NULL,
    RT_NULL,
    rt_watchdog_control,
};

rt_err_t rt_hw_watchdog_register(struct rt_watchdog_device *wtd,
                                 const char             *name,
                                 rt_uint32_t              flag,
                                 void                     *data)
{
    struct rt_device *device;
    RT_ASSERT(wtd != RT_NULL);

    device = &(wtd->parent);

    device->type      = RT_Device_Class_Miscellaneous;
    device->rx_indicate = RT_NULL;
    device->tx_complete = RT_NULL;

    device->ops       = &wdt_ops;
    device->user_data = data;

    /* register a character device */
    return rt_device_register(device, name, flag);
}
```

11.3 访问 I/O 设备

应用程序通过 I/O 设备管理接口来访问硬件设备，当设备驱动实现后，应用程序就可以访问该硬件。I/O 设备管理接口与 I/O 设备的操作方法的映射关系下图所示：

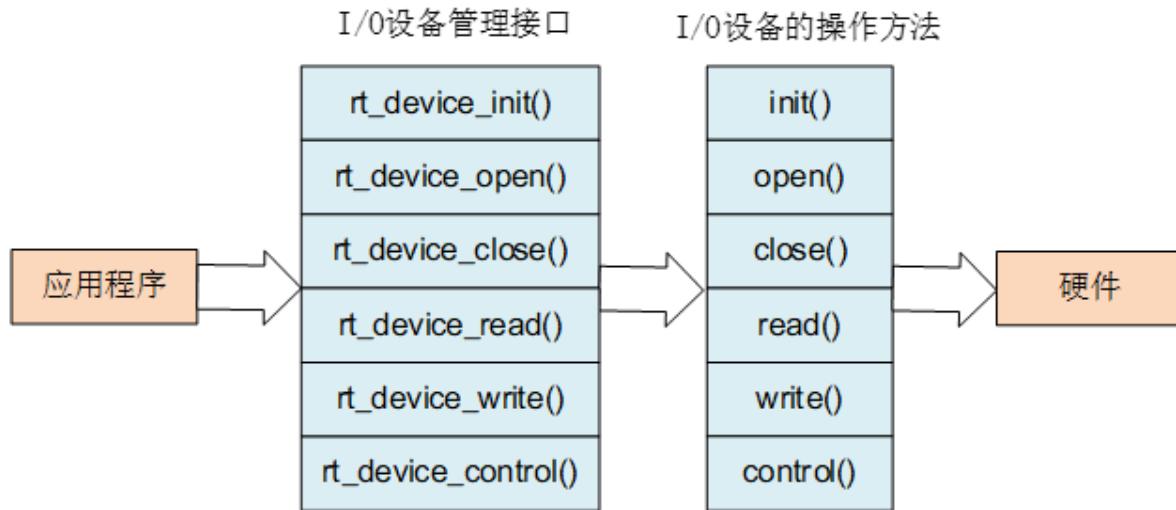


图 11.6: I/O 设备管理接口与 I/O 设备的操作方法的映射关系

11.3.1 查找设备

应用程序根据设备名称获取设备句柄，进而可以操作设备。查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	设备名称
返回	—
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

11.3.2 初始化设备

获得设备句柄后，应用程序可使用如下函数对设备进行初始化操作：

```
rt_err_t rt_device_init(rt_device_t dev);
```

参数	描述
dev	设备句柄
返回	—
RT_EOK	设备初始化成功
错误码	设备初始化失败

!!! note “注意事项” 当一个设备已经初始化成功后，调用这个接口将不再重复做初始化 0。

11.3.3 打开和关闭设备

通过设备句柄，应用程序可以打开和关闭设备，打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

```
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
```

参数	描述
dev	设备句柄
oflags	设备打开模式标志
返回	—
RT_EOK	设备打开成功
-RT_EBUSY	如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数，此设备将不允许重复打开
其他错误码	设备打开失败

oflags 支持以下的参数：

```
#define RT_DEVICE_OFLAG_CLOSE 0x000 /* 设备已经关闭（内部使用）*/
#define RT_DEVICE_OFLAG_RDONLY 0x001 /* 以只读方式打开设备 */
#define RT_DEVICE_OFLAG_WRONLY 0x002 /* 以只写方式打开设备 */
#define RT_DEVICE_OFLAG_RDWR 0x003 /* 以读写方式打开设备 */
#define RT_DEVICE_OFLAG_OPEN 0x008 /* 设备已经打开（内部使用）*/
#define RT_DEVICE_FLAG_STREAM 0x040 /* 设备以流模式打开 */
#define RT_DEVICE_FLAG_INT_RX 0x100 /* 设备以中断接收模式打开 */
#define RT_DEVICE_FLAG_DMA_RX 0x200 /* 设备以 DMA 接收模式打开 */
#define RT_DEVICE_FLAG_INT_TX 0x400 /* 设备以中断发送模式打开 */
#define RT_DEVICE_FLAG_DMA_TX 0x800 /* 设备以 DMA 发送模式打开 */
```

!!! note “注意事项” 如果上层应用程序需要设置设备的接收回调函数，则必须以 RT_DEVICE_FLAG_INT_RX 或者 RT_DEVICE_FLAG_DMA_RX 的方式打开设备，否则不会回调函数。

应用程序打开设备完成读写等操作后，如果不需要再对设备进行操作则可以关闭设备，通过如下函数完成：

```
rt_err_t rt_device_close(rt_device_t dev);
```

参数	描述
dev	设备句柄
返回	—
RT_EOK	关闭设备成功

参数	描述
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

!!! note “注意事项” 关闭设备接口和打开设备接口需配对使用，打开一次设备对应要关闭一次设备，这样设备才会被完全关闭，否则设备仍处于未关闭状态。

11.3.4 控制设备

通过命令控制字，应用程序也可以对设备进行控制，通过如下函数完成：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
```

参数	描述
dev	设备句柄
cmd	命令控制字，这个参数通常与设备驱动程序相关
arg	控制的参数
返回	—
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

参数 cmd 的通用设备命令可取如下宏定义：

#define RT_DEVICE_CTRL_RESUME	0x01	/* 恢复设备 */
#define RT_DEVICE_CTRL_SUSPEND	0x02	/* 挂起设备 */
#define RT_DEVICE_CTRL_CONFIG	0x03	/* 配置设备 */
#define RT_DEVICE_CTRL_SET_INT	0x10	/* 设置中断 */
#define RT_DEVICE_CTRL_CLR_INT	0x11	/* 清中断 */
#define RT_DEVICE_CTRL_GET_INT	0x12	/* 获取中断状态 */

11.3.5 读写设备

应用程序从设备中读取数据可以通过如下函数完成：

```
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
;
```

参数	描述
dev	设备句柄
pos	读取数据偏移量
buffer	内存缓冲区指针，读取的数据将会被保存在缓冲区中
size	读取数据的大小
返回	—
读到数据的实际大小	如果是字符设备，返回大小以字节为单位，如果是块设备，返回的大小以块为单位
0	需要读取当前线程的 <code>errno</code> 来判断错误状态

调用这个函数，会从 `dev` 设备中读取数据，并存放在 `buffer` 缓冲区中，这个缓冲区的最大长度是 `size`，`pos` 根据不同的设备类别有不同的意义。

向设备中写入数据，可以通过如下函数完成：

```
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer,
    rt_size_t size);
```

参数	描述
dev	设备句柄
pos	写入数据偏移量
buffer	内存缓冲区指针，放置要写入的数据
size	写入数据的大小
返回	—
写入数据的实际大小	如果是字符设备，返回大小以字节为单位；如果是块设备，返回的大小以块为单位
0	需要读取当前线程的 <code>errno</code> 来判断错误状态

调用这个函数，会把缓冲区 `buffer` 中的数据写入到设备 `dev` 中，写入数据的最大长度是 `size`，`pos` 根据不同的设备类别存在不同的意义。

11.3.6 数据收发回调

当硬件设备收到数据时，可以通过如下函数回调另一个函数来设置数据接收指示，通知上层应用线程有数据到达：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t
    dev, rt_size_t size));
```

参数	描述
dev	设备句柄
rx_ind	回调函数指针
返回	—
RT_EOK	设置成功

该函数的回调函数由调用者提供。当硬件设备接收到数据时，会回调这个函数并把收到的数据长度放在 size 参数中传递给上层应用。上层应用线程应在收到指示后，立刻从设备中读取数据。

在应用程序调用 `rt_device_write()` 入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件数据发送完成后（例如 DMA 传送完成或 FIFO 已经写入完毕产生完成中断时）调用。可以通过如下函数设置设备发送完成指示，函数参数及返回值见：

```
rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buffer));
```

参数	描述
dev	设备句柄
tx_done	回调函数指针
返回	—
RT_EOK	设置成功

调用这个函数时，回调函数由调用者提供，当硬件设备发送完数据时，由驱动程序回调这个函数并把发送完成的数据块地址 buffer 作为参数传递给上层应用。上层应用（线程）在收到指示时会根据发送 buffer 的情况，释放 buffer 内存块或将其作为下一个写数据的缓存。

11.3.7 设备访问示例

下面代码为用程序访问设备的示例，首先通过 `rt_device_find()` 口查找到看门狗设备，获得设备句柄，然后通过 `rt_device_init()` 口初始化设备，通过 `rt_device_control()` 口设置看门狗设备溢出时间。

```
#include <rtthread.h>
#include <rtdevice.h>

#define IWDG_DEVICE_NAME    "iwg"

static rt_device_t wdg_dev;

static void idle_hook(void)
{
    /* 在空闲线程的回调函数里喂狗 */
}
```

```
rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
rt_kprintf("feed the dog!\n");
}

int main(void)
{
    rt_err_t res = RT_EOK;
    rt_uint32_t timeout = 1000;      /* 溢出时间 */

    /* 根据设备名称查找看门狗设备，获取设备句柄 */
    wdg_dev = rt_device_find(IWDG_DEVICE_NAME);
    if (!wdg_dev)
    {
        rt_kprintf("find %s failed!\n", IWDG_DEVICE_NAME);
        return RT_ERROR;
    }
    /* 初始化设备 */
    res = rt_device_init(wdg_dev);
    if (res != RT_EOK)
    {
        rt_kprintf("initialize %s failed!\n", IWDG_DEVICE_NAME);
        return res;
    }
    /* 设置看门狗溢出时间 */
    res = rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_SET_TIMEOUT, &timeout);
    if (res != RT_EOK)
    {
        rt_kprintf("set %s timeout failed!\n", IWDG_DEVICE_NAME);
        return res;
    }
    /* 设置空闲线程回调函数 */
    rt_thread_idle_sethook(idle_hook);

    return res;
}
```

第 12 章

UART 设备

12.1 UART 简介

UART (Universal Asynchronous Receiver/Transmitter) 通用异步收发传输器，UART 作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位地传输。是在应用程序开发过程中使用频率最高的数据总线。

UART 串口的特点是将数据一位一位地顺序传送，只要 2 根传输线就可以实现双向通信，一根线发送数据的同时用另一根线接收数据。UART 串口通信有几个重要的参数，分别是波特率、起始位、数据位、停止位和奇偶检验位，对于两个使用 UART 串口通信的端口，这些参数必须匹配，否则通信将无法正常完成。UART 串口传输的数据格式如下图所示：



图 12.1：串口传输数据格式

- **起始位**：表示数据传输的开始，电平逻辑为“0”。
- **数据位**：可能值有 5、6、7、8、9，表示传输这几个 bit 位数据。一般取值为 8，因为一个 ASCII 字符值为 8 位。
- **奇偶校验位**：用于接收方对接收到的数据进行校验，校验“1”的位数为偶数(偶校验)或奇数(奇校验)，以此来校验数据传送的正确性，使用时不需要此位也可以。
- **停止位**：表示一帧数据的结束。电平逻辑为“1”。
- **波特率**：串口通信时的速率，它用单位时间内传输的二进制代码的有效位(bit)数来表示，其单位为每秒比特数 bit/s(bps)。常见的波特率值有 4800、9600、14400、38400、115200 等，数值越大数据传输的越快，波特率为 115200 表示每秒钟传输 115200 位数据。

12.2 访问串口设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问串口硬件，相关接口如下所示：

函数	描述
rt_device_find()	查找设备
rt_device_open()	打开设备
rt_device_read()	读取数据
rt_device_write()	写入数据
rt_device_control()	控制设备
rt_device_set_rx_indicate()	设置接收回调函数
rt_device_set_tx_complete()	设置发送完成回调函数
rt_device_close()	关闭设备

12.2.1 查找串口设备

应用程序根据串口设备名称获取设备句柄，进而可以操作串口设备，查找设备函数如下所示，

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

一般情况下，注册到系统的串口设备名称为 uart0, uart1 等，使用示例如下所示：

```
#define SAMPLE_UART_NAME      "uart2"    /* 串口设备名称 */
static rt_device_t serial;                  /* 串口设备句柄 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);
```

12.2.2 打开串口设备

通过设备句柄，应用程序可以打开和关闭设备，打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

```
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
```

参数	描述
dev	设备句柄
oflags	设备模式标志
返回	—
RT_EOK	设备打开成功
-RT_EBUSY	如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数，此设备将不允许重复打开
其他错误码	设备打开失败

oflags 参数支持下列取值 (可以采用或的方式支持多种取值):

#define RT_DEVICE_FLAG_STREAM	0x040	/* 流模式 */
/* 接收模式参数 */		
#define RT_DEVICE_FLAG_INT_RX	0x100	/* 中断接收模式 */
#define RT_DEVICE_FLAG_DMA_RX	0x200	/* DMA 接收模式 */
/* 发送模式参数 */		
#define RT_DEVICE_FLAG_INT_TX	0x400	/* 中断发送模式 */
#define RT_DEVICE_FLAG_DMA_TX	0x800	/* DMA 发送模式 */

串口数据接收和发送数据的模式分为 3 种：中断模式、轮询模式、DMA 模式。在使用的时候，这 3 种模式只能选其一，若串口的打开参数 oflags 没有指定使用中断模式或者 DMA 模式，则默认使用轮询模式。

DMA (Direct Memory Access) 即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过 DMA 控制器为 RAM 与 I/O 设备开辟一条直接传送数据的通路，这就节省了 CPU 的资源来做其他操作。使用 DMA 传输可以连续获取或发送一段信息而不占用中断或延时，在通信频繁或有大段信息要传输时非常有用。

!!! note “注意事项” * RT_DEVICE_FLAG_STREAM: 流模式用于向串口终端输出字符串：当输出的字符是 "\n" (对应 16 进制值为 0xA) 时，自动在前面输出一个 "\r" (对应 16 进制值为 0xD) 做分行。

流模式 RT_DEVICE_FLAG_STREAM 可以和接收发送模式参数使用或 “|” 运算符一起使用。

以中断接收及轮询发送模式使用串口设备的示例如下所示：

```
#define SAMPLE_UART_NAME      "uart2"    /* 串口设备名称 */
static rt_device_t serial;                  /* 串口设备句柄 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
```

若串口要使用 DMA 接收模式，oflags 取值 RT_DEVICE_FLAG_DMA_RX。以 DMA 接收及轮询发送模式使用串口设备的示例如下所示：

```
#define SAMPLE_UART_NAME      "uart2"    /* 串口设备名称 */
static rt_device_t serial;                  /* 串口设备句柄 */
```

```
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以 DMA 接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_DMA_RX);
```

12.2.3 控制串口设备

通过控制接口，应用程序可以对串口设备进行配置，如波特率、数据位、校验位、接收缓冲区大小、停止位等参数的修改。控制函数如下所示：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
```

参数	描述
dev	设备句柄
cmd	命令控制字，可取值：RT_DEVICE_CTRL_CONFIG
arg	控制的参数，可取类型：struct serial_configure
返回	—
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

控制参数结构体 struct serial_configure 原型如下：

```
struct serial_configure
{
    rt_uint32_t baud_rate;           /* 波特率 */
    rt_uint32_t data_bits :4;        /* 数据位 */
    rt_uint32_t stop_bits :2;        /* 停止位 */
    rt_uint32_t parity :2;          /* 奇偶校验位 */
    rt_uint32_t bit_order :1;        /* 高位在前或者低位在前 */
    rt_uint32_t invert :1;           /* 模式 */
    rt_uint32_t bufsz :16;          /* 接收数据缓冲区大小 */
    rt_uint32_t reserved :4;         /* 保留位 */
};
```

RT-Thread 提供的配置参数可取值为如下宏定义：

```
/* 波特率可取值 */
#define BAUD_RATE_2400            2400
#define BAUD_RATE_4800            4800
#define BAUD_RATE_9600            9600
#define BAUD_RATE_19200           19200
#define BAUD_RATE_38400           38400
```

```

#define BAUD_RATE_57600          57600
#define BAUD_RATE_115200         115200
#define BAUD_RATE_230400         230400
#define BAUD_RATE_460800         460800
#define BAUD_RATE_921600         921600
#define BAUD_RATE_2000000        2000000
#define BAUD_RATE_3000000        3000000
/* 数据位可取值 */
#define DATA_BITS_5               5
#define DATA_BITS_6               6
#define DATA_BITS_7               7
#define DATA_BITS_8               8
#define DATA_BITS_9               9
/* 停止位可取值 */
#define STOP_BITS_1                0
#define STOP_BITS_2                1
#define STOP_BITS_3                2
#define STOP_BITS_4                3
/* 极性位可取值 */
#define PARITY_NONE                 0
#define PARITY_ODD                  1
#define PARITY_EVEN                 2
/* 高低位顺序可取值 */
#define BIT_ORDER_LSB                 0
#define BIT_ORDER_MSB                 1
/* 模式可取值 */
#define NRZ_NORMAL                  0      /* normal mode */
#define NRZ_INVERTED                 1      /* inverted mode */
/* 接收数据缓冲区默认大小 */
#define RT_SERIAL_RB_BUFSZ            64

```

接收缓冲区：当串口使用中断接收模式打开时，串口驱动框架会根据 RT_SERIAL_RB_BUFSZ 大小开辟一块缓冲区用于保存接收到的数据，底层驱动接收到一个数据，都会在中断服务程序里面将数据放入缓冲区。

RT-Thread 提供的默认串口配置如下，即 RT-Thread 系统中默认每个串口设备都使用如下配置：

```

#define RT_SERIAL_CONFIG_DEFAULT      \
{                                         \
    BAUD_RATE_115200, /* 115200 bits/s */ \
    DATA_BITS_8,       /* 8 databits */      \
    STOP_BITS_1,       /* 1 stopbit */       \
    PARITY_NONE,       /* No parity */       \
    BIT_ORDER_LSB,     /* LSB first sent */ \
    NRZ_NORMAL,        /* Normal mode */      \
    RT_SERIAL_RB_BUFSZ, /* Buffer size */   \
    0
}

```

!!! note “注意事项” 默认串口配置接收数据缓冲区大小为 RT_SERIAL_RB_BUFSZ，即 64 字节。若一

次性数据接收字节数很多，没有及时读取数据，那么缓冲区的数据将会被新接收到的数据覆盖，造成数据丢失，建议调大缓冲区，即通过 control 接口修改。在修改缓冲区大小时请注意，缓冲区大小无法动态改变，只有在 open 设备之前可以配置。open 设备之后，缓冲区大小不可再进行更改。但除过缓冲区之外的其他参数，在 open 设备前 / 后，均可进行更改。

若实际使用串口的配置参数与默认配置参数不符，则用户可以通过应用代码进行修改。修改串口配置参数，如波特率、数据位、校验位、缓冲区接收 bufsize、停止位等的示例程序如下：

```
#define SAMPLE_UART_NAME      "uart2"      /* 串口设备名称 */
static rt_device_t serial;                      /* 串口设备句柄 */
struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT; /* 初始化配置参数 */

/* step1: 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* step2: 修改串口配置参数 */
config.baud_rate = BAUD_RATE_9600;           // 修改波特率为 9600
config.data_bits = DATA_BITS_8;                // 数据位 8
config.stop_bits = STOP_BITS_1;                // 停止位 1
config.bufsz     = 128;                        // 修改缓冲区 buff size 为 128
config.parity    = PARITY_NONE;                // 无奇偶校验位

/* step3: 控制串口设备。通过控制接口传入命令控制字，与控制参数 */
rt_device_control(serial, RT_DEVICE_CTRL_CONFIG, &config);

/* step4: 打开串口设备。以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
```

12.2.4 发送数据

向串口中写入数据，可以通过如下函数完成：

```
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer,
                          rt_size_t size);
```

参数	描述
dev	设备句柄
pos	写入数据偏移量，此参数串口设备未使用
buffer	内存缓冲区指针，放置要写入的数据
size	写入数据的大小
返回	—
写入数据的实际大小	如果是字符设备，返回大小以字节为单位；
0	需要读取当前线程的 errno 来判断错误状态

调用这个函数，会把缓冲区 buffer 中的数据写入到设备 dev 中，写入数据的大小是 size。

向串口写入数据示例程序如下所示：

```
#define SAMPLE_UART_NAME      "uart2"    /* 串口设备名称 */
static rt_device_t serial;                 /* 串口设备句柄 */
char str[] = "hello RT-Thread!\r\n";
struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT; /* 配置参数 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
/* 发送字符串 */
rt_device_write(serial, 0, str, (sizeof(str) - 1));
```

12.2.5 设置发送完成回调函数

在应用程序调用 `rt_device_write()` 写入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件数据发送完成后（例如 DMA 传送完成或 FIFO 已经写入完毕产生完成中断时）调用。可以通过如下函数设置设备发送完成指示：

```
rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buffer));
```

参数	描述
dev	设备句柄
tx_done	回调函数指针
返回	—
RT_EOK	设置成功

调用这个函数时，回调函数由调用者提供，当硬件设备发送完数据时，由设备驱动程序回调这个函数，并把发送完成的数据块地址 `buffer` 作为参数传递给上层应用。上层应用（线程）在收到指示时会根据发送 `buffer` 的情况，释放 `buffer` 内存块或将其作为下一个写数据的缓存。

12.2.6 设置接回调函数

可以通过如下函数来设置数据接收指示，当串口收到数据时，通知上层应用线程有数据到达：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev, rt_size_t size));
```

参数	描述
dev	设备句柄
rx_ind	回调函数指针
dev	设备句柄（回调函数参数）
size	缓冲区数据大小（回调函数参数）
返回	—
RT_EOK	设置成功

该函数的回调函数由调用者提供。若串口以中断接收模式打开，当串口接收到一个数据产生中断时，就会调用回调函数，并且会把此时缓冲区的数据大小放在 size 参数里，把串口设备句柄放在 dev 参数里供调用者获取。

若串口以 DMA 接收模式打开，当 DMA 完成一批数据的接收后会调用此回调函数。

一般情况下接收回调函数可以发送一个信号量或者事件通知串口数据处理线程有数据到达。使用示例如下所示：

```
#define SAMPLE_UART_NAME      "uart2"      /* 串口设备名称 */
static rt_device_t serial;                  /* 串口设备句柄 */
static struct rt_semaphore rx_sem;          /* 用于接收消息的信号量 */

/* 接收数据回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    /* 串口接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    rt_sem_release(&rx_sem);

    return RT_EOK;
}

static int uart_sample(int argc, char *argv[])
{
    serial = rt_device_find(SAMPLE_UART_NAME);

    /* 以中断接收及轮询发送模式打开串口设备 */
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);

    /* 初始化信号量 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);

    /* 设置接收回调函数 */
    rt_device_set_rx_indicate(serial, uart_input);
}
```

12.2.7 接收数据

可调用如下函数读取串口接收到的数据：

```
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size
);
```

参数	描述
dev	设备句柄
pos	读取数据偏移量，此参数串口设备未使用
buffer	缓冲区指针，读取的数据将会被保存在缓冲区中
size	读取数据的大小
返回	—
读到数据的实际大小	如果是字符设备，返回大小以字节为单位
0	需要读取当前线程的 errno 来判断错误状态

读取数据偏移量 pos 针对字符设备无效，此参数主要用于块设备中。

串口使用中断接收模式并配合接收回调函数的使用示例如下所示：

```
static rt_device_t serial; /* 串口设备句柄 */
static struct rt_semaphore rx_sem; /* 用于接收消息的信号量 */

/* 接收数据的线程 */
static void serial_thread_entry(void *parameter)
{
    char ch;

    while (1)
    {
        /* 从串口读取一个字节的数据，没有读取到则等待接收信号量 */
        while (rt_device_read(serial, -1, &ch, 1) != 1)
        {
            /* 阻塞等待接收信号量，等到信号量后再次读取数据 */
            rt_sem_take(&rx_sem, RT_WAITING_FOREVER);
        }

        /* 读取到的数据通过串口错位输出 */
        ch = ch + 1;
        rt_device_write(serial, 0, &ch, 1);
    }
}
```

12.2.8 关闭串口设备

当应用程序完成串口操作后，可以关闭串口设备，通过如下函数完成：

```
rt_err_t rt_device_close(rt_device_t dev);
```

参数	描述
dev	设备句柄
返回	—
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

关闭设备接口和打开设备接口需配对使用，打开一次设备对应要关闭一次设备，这样设备才会被完全关闭，否则设备仍处于未关闭状态。

12.3 串口设备使用示例

12.3.1 中断接收及轮询发送

示例代码的主要步骤如下所示：

- 首先查找串口设备获取设备句柄。
 - 初始化回调函数发送使用的信号量，然后以读写及中断接收方式打开串口设备。
 - 设置串口设备的接收回调函数，之后发送字符串，并创建读取数据线程。
- 读取数据线程会尝试读取一个字符数据，如果没有数据则会挂起并等待信号量，当串口设备接收到一个数据时会触发中断并调用接收回调函数，此函数会发送信号量唤醒线程，此时线程会马上读取接收到的数据。
 - 此示例代码不局限于特定的 BSP，根据 BSP 注册的串口设备，修改示例代码宏定义 SAMPLE_UART_NAME 对应的串口设备名称即可运行。

运行序列图如下图所示：

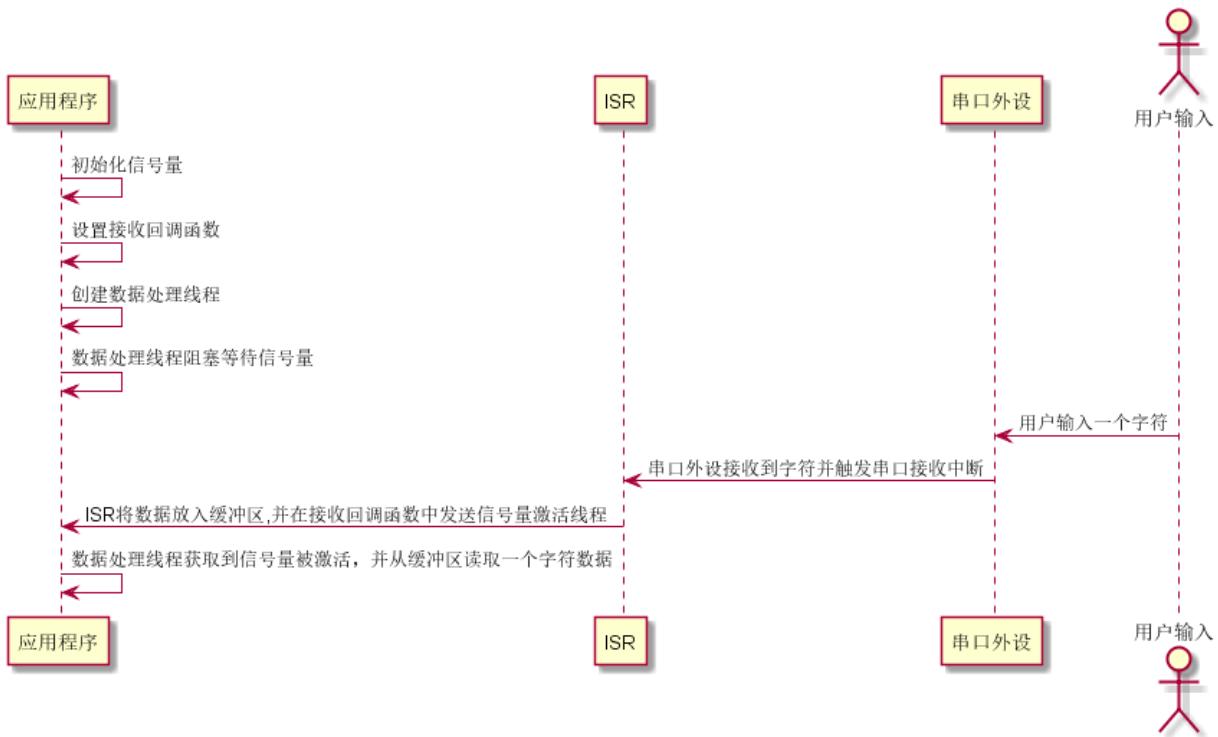


图 12.2: 串口中断接收及轮询发送序列图

```

/*
 * 程序清单：这是一个 串口 设备使用例程
 * 例程导出了 uart_sample 命令到控制终端
 * 命令调用格式：uart_sample uart2
 * 命令解释：命令第二个参数是要使用的串口设备名称，为空则使用默认的串口设备
 * 程序功能：通过串口输出字符串"hello RT-Thread!"，然后错位输出输入的字符
 */

#include <rtthread.h>

#define SAMPLE_UART_NAME          "uart2"

/* 用于接收消息的信号量 */
static struct rt_semaphore rx_sem;
static rt_device_t serial;

/* 接收数据回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    /* 串口接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    rt_sem_release(&rx_sem);

    return RT_EOK;
}

static void serial_thread_entry(void *parameter)

```

```
{  
    char ch;  
  
    while (1)  
    {  
        /* 从串口读取一个字节的数据，没有读取到则等待接收信号量 */  
        while (rt_device_read(serial, -1, &ch, 1) != 1)  
        {  
            /* 阻塞等待接收信号量，等到信号量后再次读取数据 */  
            rt_sem_take(&rx_sem, RT_WAITING_FOREVER);  
        }  
        /* 读取到的数据通过串口错位输出 */  
        ch = ch + 1;  
        rt_device_write(serial, 0, &ch, 1);  
    }  
}  
  
static int uart_sample(int argc, char *argv[])  
{  
    rt_err_t ret = RT_EOK;  
    char uart_name[RT_NAME_MAX];  
    char str[] = "hello RT-Thread!\r\n";  
  
    if (argc == 2)  
    {  
        rt_strncpy(uart_name, argv[1], RT_NAME_MAX);  
    }  
    else  
    {  
        rt_strncpy(uart_name, SAMPLE_UART_NAME, RT_NAME_MAX);  
    }  
  
    /* 查找系统中的串口设备 */  
    serial = rt_device_find(uart_name);  
    if (!serial)  
    {  
        rt_kprintf("find %s failed!\n", uart_name);  
        return RT_ERROR;  
    }  
  
    /* 初始化信号量 */  
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);  
    /* 以中断接收及轮询发送模式打开串口设备 */  
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);  
    /* 设置接收回调函数 */  
    rt_device_set_rx_indicate(serial, uart_input);  
    /* 发送字符串 */  
    rt_device_write(serial, 0, str, (sizeof(str) - 1));  
}
```

```
/* 创建 serial 线程 */
rt_thread_t thread = rt_thread_create("serial", serial_thread_entry, RT_NULL,
    1024, 25, 10);
/* 创建成功则启动线程 */
if (thread != RT_NULL)
{
    rt_thread_startup(thread);
}
else
{
    ret = RT_ERROR;
}

return ret;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(uart_sample, uart device sample);
```

12.3.2 DMA 接收及轮询发送

当串口接收到一批数据后会调用接收回调函数，接收回调函数会把此时缓冲区的数据大小通过消息队列发送给等待的数据处理线程。线程获取到消息后被激活，并读取数据。一般情况下 DMA 接收模式会结合 DMA 接收完成中断和串口空闲中断完成数据接收。

- 此示例代码不局限于特定的 BSP，根据 BSP 注册的串口设备，修改示例代码宏定义 SAMPLE_UART_NAME 对应的串口设备名称即可运行。

运行序列图如下图所示：

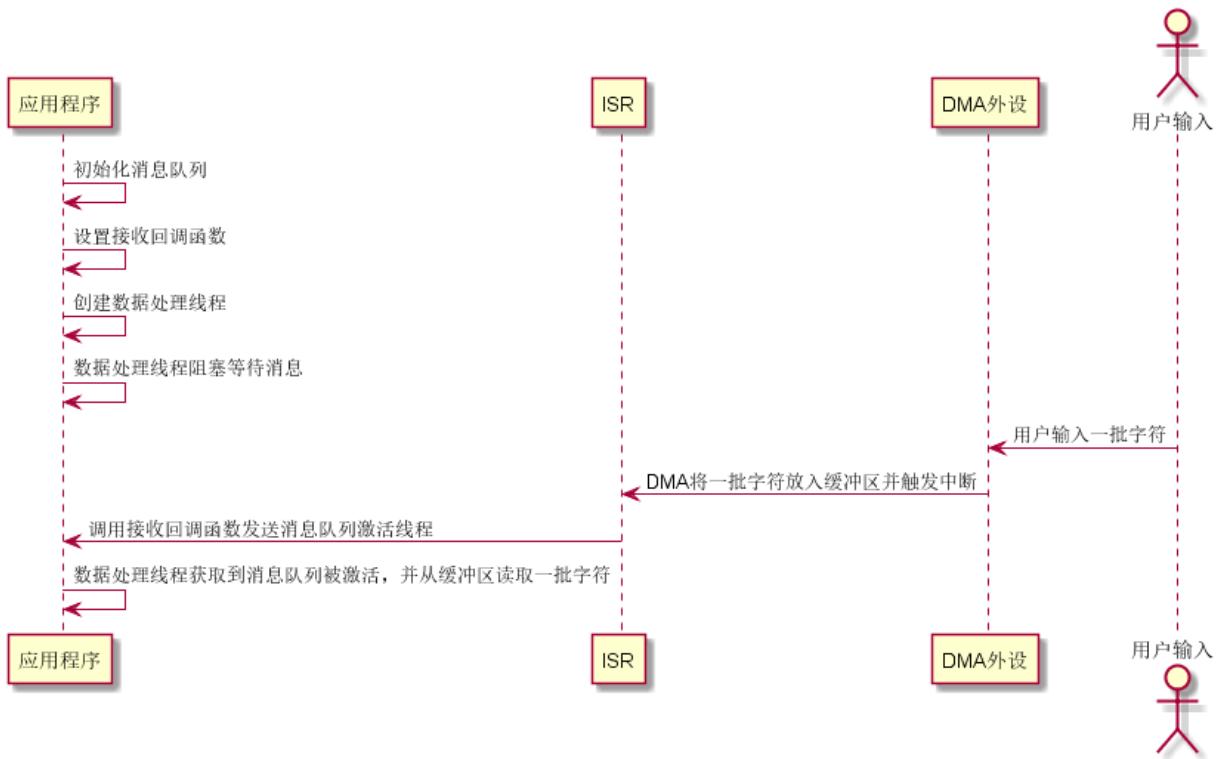


图 12.3: 串口 DMA 接收及轮询发送序列图

```

/*
 * 程序清单：这是一个串口设备 DMA 接收使用例程
 * 例程导出了 uart_dma_sample 命令到控制终端
 * 命令调用格式：uart_dma_sample uart3
 * 命令解释：命令第二个参数是要使用的串口设备名称，为空则使用默认的串口设备
 * 程序功能：通过串口输出字符串"hello RT-Thread!"，并通过串口输出接收到的数据，然后
 * 打印接收到的数据。
 */

#include <rtthread.h>

#define SAMPLE_UART_NAME          "uart3"      /* 串口设备名称 */

/* 串口接收消息结构 */
struct rx_msg
{
    rt_device_t dev;
    rt_size_t size;
};

/* 串口设备句柄 */
static rt_device_t serial;
/* 消息队列控制块 */
static struct rt_messagequeue rx_mq;

/* 接收数据回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)

```

```
{  
    struct rx_msg msg;  
    rt_err_t result;  
    msg.dev = dev;  
    msg.size = size;  
  
    result = rt_mq_send(&rx_mq, &msg, sizeof(msg));  
    if (result == -RT_EFULL)  
    {  
        /* 消息队列满 */  
        rt_kprintf("message queue full! \n");  
    }  
    return result;  
}  
  
static void serial_thread_entry(void *parameter)  
{  
    struct rx_msg msg;  
    rt_err_t result;  
    rt_uint32_t rx_length;  
    static char rx_buffer[RT_SERIAL_RB_BUFSZ + 1];  
  
    while (1)  
    {  
        rt_memset(&msg, 0, sizeof(msg));  
        /* 从消息队列中读取消息 */  
        result = rt_mq_recv(&rx_mq, &msg, sizeof(msg), RT_WAITING_FOREVER);  
        if (result == RT_EOK)  
        {  
            /* 从串口读取数据 */  
            rx_length = rt_device_read(msg.dev, 0, rx_buffer, msg.size);  
            rx_buffer[rx_length] = '\0';  
            /* 通过串口设备 serial 输出读取到的消息 */  
            rt_device_write(serial, 0, rx_buffer, rx_length);  
            /* 打印数据 */  
            rt_kprintf("%s\n", rx_buffer);  
        }  
    }  
}  
  
static int uart_dma_sample(int argc, char *argv[])  
{  
    rt_err_t ret = RT_EOK;  
    char uart_name[RT_NAME_MAX];  
    static char msg_pool[256];  
    char str[] = "hello RT-Thread!\r\n";  
  
    if (argc == 2)  
    {  
}
```

```
    rt_strncpy(uart_name, argv[1], RT_NAME_MAX);
}
else
{
    rt_strncpy(uart_name, SAMPLE_UART_NAME, RT_NAME_MAX);
}

/* 查找串口设备 */
serial = rt_device_find(uart_name);
if (!serial)
{
    rt_kprintf("find %s failed!\n", uart_name);
    return RT_ERROR;
}

/* 初始化消息队列 */
rt_mq_init(&rx_mq, "rx_mq",
           msg_pool,           /* 存放消息的缓冲区 */
           sizeof(struct rx_msg), /* 一条消息的最大长度 */
           sizeof(msg_pool),   /* 存放消息的缓冲区大小 */
           RT_IPC_FLAG_FIFO); /* 如果有多个线程等待，按照先来先得到的方法
分配消息 */

/* 以 DMA 接收及轮询发送方式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_DMA_RX);
/* 设置接收回调函数 */
rt_device_set_rx_indicate(serial, uart_input);
/* 发送字符串 */
rt_device_write(serial, 0, str, (sizeof(str) - 1));

/* 创建 serial 线程 */
rt_thread_t thread = rt_thread_create("serial", serial_thread_entry, RT_NULL,
                                       1024, 25, 10);
/* 创建成功则启动线程 */
if (thread != RT_NULL)
{
    rt_thread_startup(thread);
}
else
{
    ret = RT_ERROR;
}

return ret;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(uart_dma_sample, uart device dma sample);
```

12.3.3 串口接收不定长数据

串口接收不定长数据需要用户在应用层进行处理，一般会有特定的协议，比如一帧数据可能会有起始标记位、数据长度位、数据、终止标记位等，发送数据帧时按照约定的协议进行发送，接收数据时再按照协议进行解析。

以下是一个简单的串口接收不定长数据示例代码，仅做了数据的结束标志位 DATA_CMD_END，如果遇到结束标志，则表示一帧数据结束。示例代码的主要步骤如下所示：

1. 首先查找串口设备获取设备句柄。
 2. 初始化回调函数发送使用的信号量，然后以读写及中断接收方式打开串口设备。
 3. 设置串口设备的接回调函数，之后发送字符串，并创建解析数据线程。
- 解析数据线程会尝试读取一个字符数据，如果没有数据则会挂起并等待信号量，当串口设备接收到一个数据时会触发中断并调用接回调函数，此函数会发送信号量唤醒线程，此时线程会马上读取接收到的数据。在解析数据时，判断结束符，如果结束，则打印数据。
 - 此示例代码不局限于特定的 BSP，根据 BSP 注册的串口设备，修改示例代码宏定义 SAMPLE_UART_NAME 对应的串口设备名称即可运行。
 - 当一帧数据长度超过最大长度时，这将是一帧不合格的数据，因为后面接收到的字符将覆盖最后一个字符。

```
/*
 * 程序清单：这是一个串口设备接收不定长数据的示例代码
 * 例程导出了 uart_dma_sample 命令到控制终端
 * 命令调用格式：uart_dma_sample uart2
 * 命令解释：命令第二个参数是要使用的串口设备名称，为空则使用默认的串口设备
 * 程序功能：通过串口 uart2 输出字符串"hello RT-Thread!"，并通过串口 uart2 输入一串
 *             字符（不定长），再通过数据解析后，使用控制台显示有效数据。
 */

#include <rtthread.h>

#define SAMPLE_UART_NAME          "uart2"
#define DATA_CMD_END              '\r'      /* 结束位设置为 \r，即回车符 */
#define ONE_DATA_MAXLEN           20       /* 不定长数据的最大长度 */

/* 用于接收消息的信号量 */
static struct rt_semaphore rx_sem;
static rt_device_t serial;

/* 接收数据回调函数 */
static rt_err_t uart_rx_ind(rt_device_t dev, rt_size_t size)
{
    /* 串口接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    if (size > 0)
    {
        rt_sem_release(&rx_sem);
    }
}
```

```
        return RT_EOK;
    }

static char uart_sample_get_char(void)
{
    char ch;

    while (rt_device_read(serial, 0, &ch, 1) == 0)
    {
        rt_sem_control(&rx_sem, RT_IPC_CMD_RESET, RT_NULL);
        rt_sem_take(&rx_sem, RT_WAITING_FOREVER);
    }
    return ch;
}

/* 数据解析线程 */
static void data_parsing(void)
{
    char ch;
    char data[ONE_DATA_MAXLEN];
    static char i = 0;

    while (1)
    {
        ch = uart_sample_get_char();
        rt_device_write(serial, 0, &ch, 1);
        if(ch == DATA_CMD_END)
        {
            data[i++] = '\0';
            rt_kprintf("data=%s\r\n",data);
            i = 0;
            continue;
        }
        i = (i >= ONE_DATA_MAXLEN-1) ? ONE_DATA_MAXLEN-1 : i;
        data[i++] = ch;
    }
}

static int uart_data_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    char uart_name[RT_NAME_MAX];
    char str[] = "hello RT-Thread!\r\n";

    if (argc == 2)
    {
        rt_strncpy(uart_name, argv[1], RT_NAME_MAX);
    }
    else
```

```
{  
    rt_strncpy(uart_name, SAMPLE_UART_NAME, RT_NAME_MAX);  
}  
  
/* 查找系统中的串口设备 */  
serial = rt_device_find(uart_name);  
if (!serial)  
{  
    rt_kprintf("find %s failed!\n", uart_name);  
    return RT_ERROR;  
}  
  
/* 初始化信号量 */  
rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);  
/* 以中断接收及轮询发送模式打开串口设备 */  
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);  
/* 设置接收回调函数 */  
rt_device_set_rx_indicate(serial, uart_rx_ind);  
/* 发送字符串 */  
rt_device_write(serial, 0, str, (sizeof(str) - 1));  
  
/* 创建 serial 线程 */  
rt_thread_t thread = rt_thread_create("serial", (void (*)(void *))parameter)  
    data_parsing, RT_NULL, 1024, 25, 10);  
/* 创建成功则启动线程 */  
if (thread != RT_NULL)  
{  
    rt_thread_startup(thread);  
}  
else  
{  
    ret = RT_ERROR;  
}  
  
return ret;  
}  
  
/* 导出到 msh 命令列表中 */  
MSH_CMD_EXPORT(uart_data_sample, uart device sample);
```

第 13 章

PIN 设备

13.1 引脚简介

芯片上的引脚一般分为 4 类：电源、时钟、控制与 I/O，I/O 口在使用模式上又分为 General Purpose Input Output（通用输入 / 输出），简称 GPIO，与功能复用 I/O（如 SPI/I2C/UART 等）。

大多数 MCU 的引脚都不止一个功能。不同引脚内部结构不一样，拥有的功能也不一样。可以通过不同的配置，切换引脚的实际功能。通用 I/O 口主要特性如下：

- 可编程控制中断：中断触发模式可配置，一般有下图所示 5 种中断触发模式：

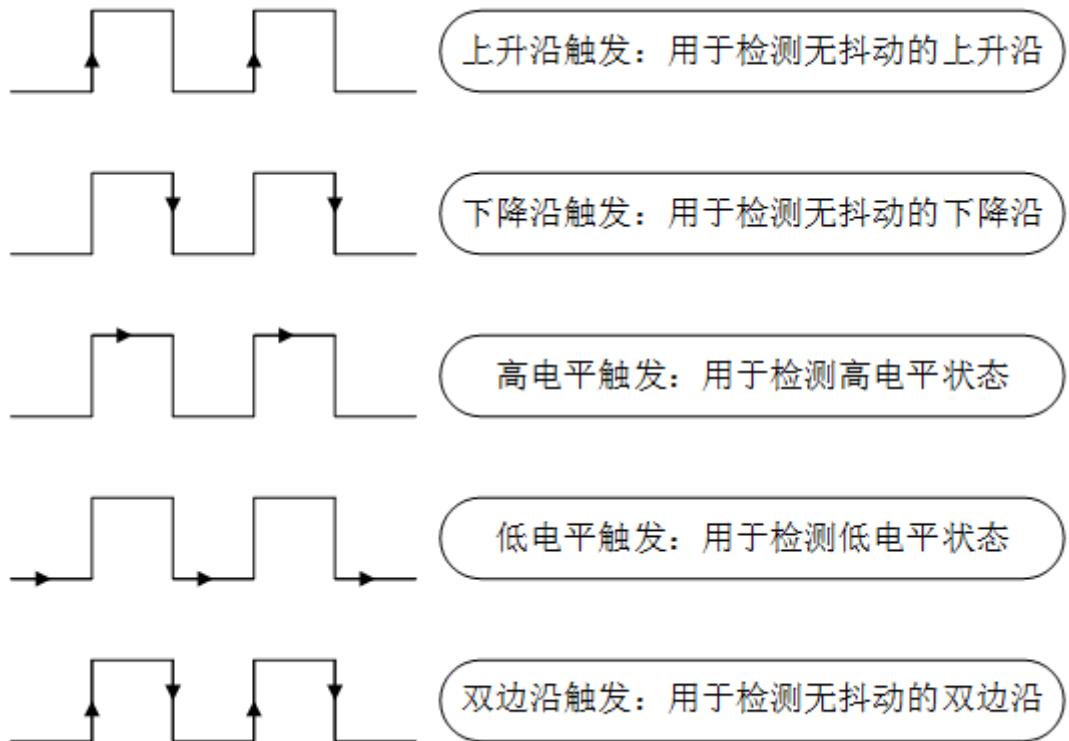


图 13.1: 5 种中断触发模式

- 输入输出模式可控制。
 - 输出模式一般包括：推挽、开漏、上拉、下拉。引脚为输出模式时，可以通过配置引脚输出的电平状态为高电平或低电平来控制连接的外围设备。
 - 输入模式一般包括：浮空、上拉、下拉、模拟。引脚为输入模式时，可以读取引脚的电平状态，即高电平或低电平。

13.2 访问 PIN 设备

应用程序通过 RT-Thread 提供的 PIN 设备管理接口来访问 GPIO，相关接口如下所示：

函数	描述
rt_pin_mode()	设置引脚模式
rt_pin_write()	设置引脚电平
rt_pin_read()	读取引脚电平
rt_pin_attach_irq()	绑定引脚中断回调函数
rt_pin_irq_enable()	使能引脚中断
rt_pin_detach_irq()	脱离引脚中断回调函数

13.2.1 获取引脚编号

RT-Thread 提供的引脚编号需要和芯片的引脚号区分开来，它们并不是同一个概念，引脚编号由 PIN 设备驱动程序定义，和具体的芯片相关。有 2 种方式可以获取引脚编号：使用宏定义或者查看 PIN 驱动文件。

13.2.1.1 使用宏定义

如果使用 `rt-thread/bsp/stm32` 目录下的 BSP 则可以使用下面的宏获取引脚编号：

```
GET_PIN(port, pin)
```

获取引脚号为 PF9 的 LED0 对应的引脚编号的示例代码如下所示：

```
#define LED0_PIN      GET_PIN(F, 9)
```

13.2.1.2 查看驱动文件

如果使用其他 BSP 则需要查看 PIN 驱动代码 `drv_gpio.c` 文件确认引脚编号。此文件里有一个数组存放了每个 PIN 脚对应的编号信息，如下所示：

```
static const rt_uint16_t pins[] =
{
```

```

__STM32_PIN_DEFAULT,
__STM32_PIN_DEFAULT,
__STM32_PIN(2, A, 15),
__STM32_PIN(3, B, 5),
__STM32_PIN(4, B, 8),
__STM32_PIN_DEFAULT,
__STM32_PIN_DEFAULT,
__STM32_PIN(8, A, 14),
__STM32_PIN(9, B, 6),
...
}

```

以`__STM32_PIN(2, A, 15)`为例，2 为 RT-Thread 使用的引脚编号，A 为端口号，15 为引脚号，所以 PA15 对应的引脚编号为 2。

13.2.2 设置引脚模式

引脚在使用前需要先设置好输入或者输出模式，通过如下函数完成：

```
void rt_pin_mode(rt_base_t pin, rt_base_t mode);
```

参数	描述
pin	引脚编号
mode	引脚工作模式

目前 RT-Thread 支持的引脚工作模式可取如所示的 5 种宏定义值之一，每种模式对应的芯片实际支持的模式需参考 PIN 设备驱动程序的具体实现：

```

#define PIN_MODE_OUTPUT 0x00          /* 输出 */
#define PIN_MODE_INPUT 0x01           /* 输入 */
#define PIN_MODE_INPUT_PULLUP 0x02    /* 上拉输入 */
#define PIN_MODE_INPUT_PULLDOWN 0x03   /* 下拉输入 */
#define PIN_MODE_OUTPUT_OD 0x04        /* 开漏输出 */

```

使用示例如下所示：

```

#define BEEP_PIN_NUM      35 /* PB0 */

/* 蜂鸣器引脚为输出模式 */
rt_pin_mode(BEEP_PIN_NUM, PIN_MODE_OUTPUT);

```

13.2.3 设置引脚电平

设置引脚输出电平的函数如下所示：

```
void rt_pin_write(rt_base_t pin, rt_base_t value);
```

参数	描述
pin	引脚编号
value	电平逻辑值，可取 2 种宏定义值之一：PIN_LOW 低电平，PIN_HIGH 高电平

使用示例如下所示：

```
#define BEEP_PIN_NUM          35 /* PB0 */

/* 蜂鸣器引脚为输出模式 */
rt_pin_mode(BEEP_PIN_NUM, PIN_MODE_OUTPUT);
/* 设置低电平 */
rt_pin_write(BEEP_PIN_NUM, PIN_LOW);
```

13.2.4 读取引脚电平

读取引脚电平的函数如下所示：

```
int rt_pin_read(rt_base_t pin);
```

参数	描述
pin	引脚编号
返回	—
PIN_LOW	低电平
PIN_HIGH	高电平

使用示例如下所示：

```
#define BEEP_PIN_NUM          35 /* PB0 */
int status;

/* 蜂鸣器引脚为输出模式 */
rt_pin_mode(BEEP_PIN_NUM, PIN_MODE_OUTPUT);
/* 设置低电平 */
rt_pin_write(BEEP_PIN_NUM, PIN_LOW);

status = rt_pin_read(BEEP_PIN_NUM);
```

13.2.5 绑定引脚中断回调函数

若要使用到引脚的中断功能，可以使用如下函数将某个引脚配置为某种中断触发模式并绑定一个中断回调函数到对应引脚，当引脚中断发生时，就会执行回调函数：

```
rt_err_t rt_pin_attach_irq(rt_int32_t pin, rt_uint32_t mode,
                           void (*hdr)(void *args), void *args);
```

参数	描述
pin	引脚编号
mode	中断触发模式
hdr	中断回调函数，用户需要自行定义这个函数
args	中断回调函数的参数，不需要时设置为 RT_NULL
返回	—
RT_EOK	绑定成功
错误码	绑定失败

中断触发模式 mode 可取如下 5 种宏定义值之一：

```
#define PIN_IRQ_MODE_RISING 0x00      /* 上升沿触发 */
#define PIN_IRQ_MODE_FALLING 0x01      /* 下降沿触发 */
#define PIN_IRQ_MODE_RISING_FALLING 0x02 /* 边沿触发（上升沿和下降沿都触发） */
#define PIN_IRQ_MODE_HIGH_LEVEL 0x03    /* 高电平触发 */
#define PIN_IRQ_MODE_LOW_LEVEL 0x04     /* 低电平触发 */
```

使用示例如下所示：

```
#define KEY0_PIN_NUM          55 /* PD8 */
/* 中断回调函数 */
void beep_on(void *args)
{
    rt_kprintf("turn on beep!\n");

    rt_pin_write(BEEP_PIN_NUM, PIN_HIGH);
}
static void pin_beep_sample(void)
{
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为beep_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, beep_on, RT_NULL);
}
```

13.2.6 使能引脚中断

绑定好引脚中断回调函数后使用下面的函数使能引脚中断：

```
rt_err_t rt_pin_irq_enable(rt_base_t pin, rt_uint32_t enabled);
```

参数	描述
pin	引脚编号
enabled	状态，可取 2 种值之一：PIN_IRQ_ENABLE（开启），PIN_IRQ_DISABLE（关闭）
返回	—
RT_EOK	使能成功
错误码	使能失败

使用示例如下所示：

```
#define KEY0_PIN_NUM          55 /* PD8 */
/* 中断回调函数 */
void beep_on(void *args)
{
    rt_kprintf("turn on beep!\n");

    rt_pin_write(BEEP_PIN_NUM, PIN_HIGH);
}
static void pin_beep_sample(void)
{
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为beep_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, beep_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);
}
```

13.2.7 脱离引脚中断回调函数

可以使用如下函数脱离引脚中断回调函数：

```
rt_err_t rt_pin_detach_irq(rt_int32_t pin);
```

参数	描述
pin	引脚编号
返回	—

参数	描述
RT_EOK	脱离成功
错误码	脱离失败

引脚脱离了中断回调函数以后，中断并没有关闭，还可以调用绑定中断回调函数再次绑定其他回调函数。

```
#define KEY0_PIN_NUM          55 /* PD8 */
/* 中断回调函数 */
void beep_on(void *args)
{
    rt_kprintf("turn on beep!\n");

    rt_pin_write(BEEP_PIN_NUM, PIN_HIGH);
}
static void pin_beep_sample(void)
{
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为beep_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, beep_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);
    /* 脱离中断回调函数 */
    rt_pin_detach_irq(KEY0_PIN_NUM);
}
```

13.3 PIN 设备使用示例

PIN 设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 设置蜂鸣器对应引脚为输出模式，并给一个默认的低电平状态。
2. 设置按键 0 和按键 1 对应引脚为输入模式，然后绑定中断回调函数并使能中断。
3. 按下按键 0 蜂鸣器开始响，按下按键 1 蜂鸣器停止响。

```
/*
 * 程序清单：这是一个 PIN 设备使用例程
 * 例程导出了 pin_beep_sample 命令到控制终端
 * 命令调用格式：pin_beep_sample
 * 程序功能：通过按键控制蜂鸣器对应引脚的电平状态控制蜂鸣器
*/
#include <rtthread.h>
```

```
#include <rtdevice.h>

/* 引脚编号，通过查看设备驱动文件drv_gpio.c确定 */
#ifndef BEEP_PIN_NUM
    #define BEEP_PIN_NUM          35 /* PB0 */
#endif
#ifndef KEY0_PIN_NUM
    #define KEY0_PIN_NUM          55 /* PD8 */
#endif
#ifndef KEY1_PIN_NUM
    #define KEY1_PIN_NUM          56 /* PD9 */
#endif

void beep_on(void *args)
{
    rt_kprintf("turn on beep!\n");

    rt_pin_write(BEEP_PIN_NUM, PIN_HIGH);
}

void beep_off(void *args)
{
    rt_kprintf("turn off beep!\n");

    rt_pin_write(BEEP_PIN_NUM, PIN_LOW);
}

static void pin_beep_sample(void)
{
    /* 蜂鸣器引脚为输出模式 */
    rt_pin_mode(BEEP_PIN_NUM, PIN_MODE_OUTPUT);
    /* 默认低电平 */
    rt_pin_write(BEEP_PIN_NUM, PIN_LOW);

    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，下降沿模式，回调函数名为beep_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, beep_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);

    /* 按键1引脚为输入模式 */
    rt_pin_mode(KEY1_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，下降沿模式，回调函数名为beep_off */
    rt_pin_attach_irq(KEY1_PIN_NUM, PIN_IRQ_MODE_FALLING, beep_off, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY1_PIN_NUM, PIN_IRQ_ENABLE);
}

/* 导出到 msh 命令列表中 */
```

```
MSH_CMD_EXPORT(pin_beep_sample, pin beep sample);
```

第 14 章

ADC 设备

14.1 ADC 简介

ADC(Analog-to-Digital Converter) 指模数转换器。是指将连续变化的模拟信号转换为离散的数字信号的器件。真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发射的数字形式。模数转换器可以实现这个功能，在各种不同的产品中都可以找到它的身影。与之相对应的 DAC(Digital-to-Analog Converter)，它是 ADC 模数转换的逆向过程。ADC 最早用于对无线信号向数字信号转换。如电视信号，长短播电台发接收等。

14.1.1 转换过程

如下图所示模数转换一般要经过采样、保持和量化、编码这几个步骤。在实际电路中，有些过程是合并进行的，如采样和保持，量化和编码在转换过程中是同时实现的。

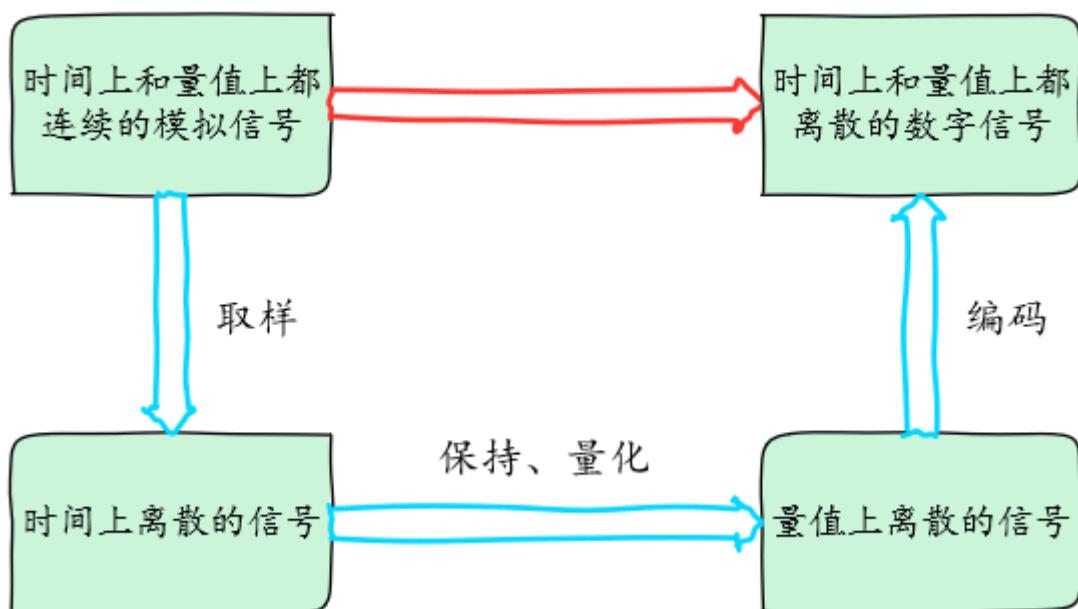


图 14.1: ADC 转换过程

采样是将时间上连续变化的模拟信号转换为时间上离散的模拟信号。采样取得的模拟信号转换为数字信号都需要一定时间，为了给后续的量化编码过程提供一个稳定的值，在采样电路后要求将所采样的模拟信号保持一段时间。

将数值连续的模拟量转换为数字量的过程称为量化。数字信号在数值上是离散的。采样保持电路的输出电压还需要按照某种近似方式归化到与之相应的离散电平上，任何数字量只能是某个最小数量单位的整数倍。量化后的数值最后还需要编码过程，也就是 A/D 转换器输出的数字量。

14.1.2 分辨率

分辨率以二进制（或十进制）数的位数来表示，一般有 8 位、10 位、12 位、16 位等，它说明模数转换器对输入信号的分辨能力，位数越多，表示分辨率越高，恢复模拟信号时会更精确。

14.1.3 精度

精度表示 ADC 器件在所有的数值点上对应的模拟值和真实值之间的最大误差值，也就是输出数值偏离线性最大的距离。

!!! note “注意事项” 精度与分辨率是两个不一样的概念，请注意区分。

14.1.4 转换速率

转换速率是指 A/D 转换器完成一次从模拟到数字的 AD 转换所需时间的倒数。例如，某 A/D 转换器的转换速率为 1MHz，则表示完成一次 AD 转换时间为 1 微秒。

14.2 访问 ADC 设备

应用程序通过 RT-Thread 提供的 ADC 设备管理接口来访问 ADC 硬件，相关接口如下所示：

函数	描述
rt_device_find()	根据 ADC 设备名称查找设备获取设备句柄
rt_adc_enable()	使能 ADC 设备
rt_adc_read()	读取 ADC 设备数据
rt_adc_disable()	关闭 ADC 设备

14.2.1 查找 ADC 设备

应用程序根据 ADC 设备名称获取设备句柄，进而可以操作 ADC 设备，查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	ADC 设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

一般情况下，注册到系统的 ADC 设备名称为 adc0, adc1 等，使用示例如下所示：

```
#define ADC_DEV_NAME      "adc1" /* ADC 设备名称 */
rt_adc_device_t adc_dev;           /* ADC 设备句柄 */
/* 查找设备 */
adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
```

14.2.2 使能 ADC 通道

在读取 ADC 设备数据前需要先使能设备，通过如下函数使能设备：

```
rt_err_t rt_adc_enable(rt_adc_device_t dev, rt_uint32_t channel);
```

参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	——
RT_EOK	成功
-RT_ENOSYS	失败，设备操作方法为空
其他错误码	失败

使用示例如下所示：

```
#define ADC_DEV_NAME      "adc1" /* ADC 设备名称 */
#define ADC_DEV_CHANNEL    5       /* ADC 通道 */
rt_adc_device_t adc_dev;           /* ADC 设备句柄 */
/* 查找设备 */
adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
/* 使能设备 */
rt_adc_enable(adc_dev, ADC_DEV_CHANNEL);
```

14.2.3 读取 ADC 通道采样值

读取 ADC 通道采样值可通过如下函数完成：

```
rt_uint32_t rt_adc_read(rt_adc_device_t dev, rt_uint32_t channel);
```

参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	—
读取的数值	

使用 ADC 采样电压值的使用示例如下所示：

```
#define ADC_DEV_NAME      "adc1" /* ADC 设备名称 */
#define ADC_DEV_CHANNEL    5       /* ADC 通道 */
#define REFER_VOLTAGE      330     /* 参考电压 3.3V, 数据精度乘以100保留2位小数
*/
#define CONVERT_BITS        (1 << 12) /* 转换位数为12位 */

rt_adc_device_t adc_dev;           /* ADC 设备句柄 */
rt_uint32_t value;
/* 查找设备 */
adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
/* 使能设备 */
rt_adc_enable(adc_dev, ADC_DEV_CHANNEL);
/* 读取采样值 */
value = rt_adc_read(adc_dev, ADC_DEV_CHANNEL);
/* 转换为对应电压值 */
vol = value * REFER_VOLTAGE / CONVERT_BITS;
rt_kprintf("the voltage is :%d.%02d \n", vol / 100, vol % 100);
```

实际电压值的计算公式为：采样值 * 参考电压 / (1 << 分辨率位数)，上面示例代码乘以 100 将数据放大，最后通过 vol / 100 获得电压的整数位值，通过 vol % 100 获得电压的小数位值。

14.2.4 关闭 ADC 通道

关闭 ADC 通道可通过如下函数完成：

```
rt_err_t rt_adc_disable(rt_adc_device_t dev, rt_uint32_t channel);
```

参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	—
RT_EOK	成功

参数	描述
-RT_ENOSYS	失败，设备操作方法为空
其他错误码	失败

使用示例如下所示：

```
#define ADC_DEV_NAME      "adc1" /* ADC 设备名称 */
#define ADC_DEV_CHANNEL    5       /* ADC 通道 */
rt_adc_device_t adc_dev;           /* ADC 设备句柄 */
rt_uint32_t value;
/* 查找设备 */
adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
/* 使能设备 */
rt_adc_enable(adc_dev, ADC_DEV_CHANNEL);
/* 读取采样值 */
value = rt_adc_read(adc_dev, ADC_DEV_CHANNEL);
/* 转换为对应电压值 */
vol = value * REFER_VOLTAGE / CONVERT_BITS;
rt_kprintf("the voltage is :%d.%02d \n", vol / 100, vol % 100);
/* 关闭通道 */
rt_adc_disable(adc_dev, ADC_DEV_CHANNEL);
```

14.2.5 FinSH 命令

在使用设备前，需要先查找设备是否存在，可以使用命令 `adc probe` 后面跟注册的 ADC 设备的名称。如下所示：

```
msh >adc probe adc1
probe adc1 success
```

使能设备的某个通道可以使用命令 `adc enable` 后面跟通道号。

```
msh >adc enable 5
adc1 channel 5 enables success
```

读取 ADC 设备某个通道的数据可以使用命令 `adc read` 后面跟通道号。

```
msh >adc read 5
adc1 channel 5  read value is 0x00000FFF
msh >
```

关闭设备的某个通道可以使用命令 `adc disable` 后面跟通道号。

```
msh >adc disable 5
adc1 channel 5 disable success
msh >
```

14.3 ADC 设备使用示例

ADC 设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 首先根据 ADC 设备名称“adc1”查找设备获取设备句柄。
2. 使能设备后读取 adc1 设备对应的通道 5 的采样值，然后根据分辨率为 12 位，参考电压为 3.3V 计算实际的电压值。
3. 最后关闭 ADC 设备对应通道。

运行结果：打印实际读取到的转换的原始数据和经过计算后的实际电压值。

```
/*
 * 程序清单： ADC 设备使用例程
 * 例程导出了 adc_sample 命令到控制终端
 * 命令调用格式：adc_sample
 * 程序功能：通过 ADC 设备采样电压值并转换为数值。
 *           示例代码参考电压为3.3V,转换位数为12位。
 */

#include <rtthread.h>
#include <rtdevice.h>

#define ADC_DEV_NAME      "adc1"        /* ADC 设备名称 */
#define ADC_DEV_CHANNEL   5             /* ADC 通道 */
#define REFER_VOLTAGE     330          /* 参考电压 3.3V, 数据精度乘以100保留2位小数
*/
#define CONVERT_BITS       (1 << 12)    /* 转换位数为12位 */

static int adc_vol_sample(int argc, char *argv[])
{
    rt_adc_device_t adc_dev;
    rt_uint32_t value, vol;
    rt_err_t ret = RT_EOK;

    /* 查找设备 */
    adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
    if (adc_dev == RT_NULL)
    {
        rt_kprintf("adc sample run failed! can't find %s device!\n", ADC_DEV_NAME);
        return RT_ERROR;
    }

    /* 使能设备 */
    ret = rt_adc_enable(adc_dev, ADC_DEV_CHANNEL);

    /* 读取采样值 */
    value = rt_adc_read(adc_dev, ADC_DEV_CHANNEL);
```

```
rt_kprintf("the value is :%d \n", value);

/* 转换为对应电压值 */
vol = value * REFER_VOLTAGE / CONVERT_BITS;
rt_kprintf("the voltage is :%d.%02d \n", vol / 100, vol % 100);

/* 关闭通道 */
ret = rt_adc_disableadc_dev, ADC_DEV_CHANNEL);

return ret;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(adc_vol_sample, adc voltage convert sample);
```

14.4 常见问题

14.4.1 Q: menuconfig 找不到 ADC 设备的配置选项？

A: 使用的源代码还不支持 ADC 设备驱动框架。建议更新源代码。

第 15 章

HWTIMER 设备

15.1 定时器简介

硬件定时器一般有 2 种工作模式，定时器模式和计数器模式。不管是工作在哪一种模式，实质都是通过内部计数器模块对脉冲信号进行计数。下面是定时器的一些重要概念。

计数器模式：对外部输入引脚的外部脉冲信号计数。

定时器模式：对内部脉冲信号计数。定时器常用作定时时钟，以实现定时检测，定时响应、定时控制。

计数器：计数器可以递增计数或者递减计数。16 位计数器的最大计数值为 65535，32 位的最大值为 4294967295。

计数频率：定时器模式时，计数器单位时间内的计数次数，由于系统时钟频率是定值，所以可以根据计数器的计数值计算出定时时间，定时时间 = 计数值 / 计数频率。例如计数频率为 1MHz，计数器计数一次的时间则为 $1 / 1000000$ ，也就是每经过 1 微秒计数器加一（或减一），此时 16 位计数器的最大定时能力为 65535 微秒，即 65.535 毫秒。

15.2 访问硬件定时器设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问硬件定时器设备，相关接口如下所示：

函数	描述
rt_device_find()	查找定时器设备
rt_device_open()	以读写方式打开定时器设备
rt_device_set_rx_indicate()	设置超时回调函数
rt_device_control()	控制定时器设备，可以设置定时模式（单次/周期）/计数频率，或者停止定时器
rt_device_write()	设置定时器超时值，定时器随即启动
rt_device_read()	获取定时器当前值
rt_device_close()	关闭定时器设备

15.2.1 查找定时器设备

应用程序根据硬件定时器设备名称获取设备句柄，进而可以操作硬件定时器设备，查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	硬件定时器设备名称
返回	——
定时器设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

一般情况下，注册到系统的硬件定时器设备名称为 timer0, timer1 等，使用示例如下所示：

```
#define HWTIMER_DEV_NAME "timer0" /* 定时器名称 */
rt_device_t hw_dev; /* 定时器设备句柄 */
/* 查找定时器设备 */
hw_dev = rt_device_find(HWTIMER_DEV_NAME);
```

15.2.2 打开定时器设备

通过设备句柄，应用程序可以打开设备。打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

```
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
```

参数	描述
dev	硬件定时器设备句柄
oflags	设备打开模式，一般以读写方式打开，即取值： RT_DEVICE_OFLAG_RDWR
返回	——
RT_EOK	设备打开成功
其他错误码	设备打开失败

使用示例如下所示：

```
#define HWTIMER_DEV_NAME "timer0" /* 定时器名称 */
rt_device_t hw_dev; /* 定时器设备句柄 */
/* 查找定时器设备 */
hw_dev = rt_device_find(HWTIMER_DEV_NAME);
```

```
/* 以读写方式打开设备 */
rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
```

15.2.3 设置超时回调函数

通过如下函数设置定时器超时回调函数，当定时器超时将会调用此回调函数：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev, rt_size_t size))
```

参数	描述
dev	设备句柄
rx_ind	超时回调函数，由调用者提供
返回	—
RT_EOK	成功

使用示例如下所示：

```
#define HWTIMER_DEV_NAME "timer0" /* 定时器名称 */
rt_device_t hw_dev; /* 定时器设备句柄 */

/* 定时器超时回调函数 */
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback function!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());

    return 0;
}

static int hwtimer_sample(int argc, char *argv[])
{
    /* 查找定时器设备 */
    hw_dev = rt_device_find(HWTIMER_DEV_NAME);
    /* 以读写方式打开设备 */
    rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
    /* 设置超时回调函数 */
    rt_device_set_rx_indicate(hw_dev, timeout_cb);
}
```

15.2.4 控制定时器设备

通过命令控制字，应用程序可以对硬件定时器设备进行配置，通过如下函数完成：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
```

参数	描述
dev	设备句柄
cmd	命令控制字
arg	控制的参数
返回	—
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败, dev 为空
其他错误码	执行失败

硬件定时器设备支持的命令控制字如下所示：

控制字	描述
HWTIMER_CTRL_FREQ_SET	设置计数频率
HWTIMER_CTRL_STOP	停止定时器
HWTIMER_CTRL_INFO_GET	获取定时器特征信息
HWTIMER_CTRL_MODE_SET	设置定时器模式

获取定时器特征信息参数 `arg` 为指向结构体 `struct rt_hwtimer_info` 的指针，作为一个输出参数保存获取的信息。

!!! note “注意事项” 定时器硬件及驱动支持设置计数频率的情况下设置频率才有效，一般使用驱动设置的默认频率即可。

设置定时器模式时，参数 `arg` 可取如下值：

HWTIMER_MODE_ONESHOT	单次定时
HWTIMER_MODE_PERIOD	周期性定时

设置定时器计数频率和定时模式的使用示例如下所示：

```
#define HWTIMER_DEV_NAME "timer0" /* 定时器名称 */
rt_device_t hw_dev; /* 定时器设备句柄 */
rt_hwtimer_mode_t mode; /* 定时器模式 */
rt_uint32_t freq = 10000; /* 计数频率 */

/* 定时器超时回调函数 */
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback function!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());
```

```

    return 0;
}

static int hwtimer_sample(int argc, char *argv[])
{
    /* 查找定时器设备 */
    hw_dev = rt_device_find(HWTIMER_DEV_NAME);
    /* 以读写方式打开设备 */
    rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
    /* 设置超时回调函数 */
    rt_device_set_rx_indicate(hw_dev, timeout_cb);

    /* 设置计数频率(默认1Mhz或支持的最小计数频率) */
    rt_device_control(hw_dev, HWTIMER_CTRL_FREQ_SET, &freq);
    /* 设置模式为周期性定时器 */
    mode = HWTIMER_MODE_PERIOD;
    rt_device_control(hw_dev, HWTIMER_CTRL_MODE_SET, &mode);
}

```

15.2.5 设置定时器超时值

通过如下函数可以设置定时器的超时值：

```
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer,
    rt_size_t size);
```

参数	描述
dev	设备句柄
pos	写入数据偏移量，未使用，可取 0 值
buffer	指向定时器超时时间结构体的指针
size	超时时间结构体的大小
返回	—
写入数据的实际大小	
0	失败

超时时间结构体原型如下所示：

```
typedef struct rt_hwtimerval
{
    rt_int32_t sec;      /* 秒 s */
    rt_int32_t usec;     /* 微秒 us */
```

```
 } rt_hwtimerval_t;
```

设置定时器超时值的使用示例如下所示：

```
#define HWTIMER_DEV_NAME "timer0"      /* 定时器名称 */
rt_device_t hw_dev;                      /* 定时器设备句柄 */
rt_hwtimer_mode_t mode;                  /* 定时器模式 */
rt_hwtimerval_t timeout_s;               /* 定时器超时值 */

/* 定时器超时回调函数 */
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback function!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());

    return 0;
}

static int hwtimer_sample(int argc, char *argv[])
{
    /* 查找定时器设备 */
    hw_dev = rt_device_find(HWTIMER_DEV_NAME);
    /* 以读写方式打开设备 */
    rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
    /* 设置超时回调函数 */
    rt_device_set_rx_indicate(hw_dev, timeout_cb);
    /* 设置模式为周期性定时器 */
    mode = HWTIMER_MODE_PERIOD;
    rt_device_control(hw_dev, HWTIMER_CTRL_MODE_SET, &mode);

    /* 设置定时器超时值为5s并启动定时器 */
    timeout_s.sec = 5;        /* 秒 */
    timeout_s.usec = 0;       /* 微秒 */
    rt_device_write(hw_dev, 0, &timeout_s, sizeof(timeout_s));
}
```

15.2.6 获取定时器当前值

通过如下函数可以获取定时器当前值：

```
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size
);
```

参数	描述
dev	定时器设备句柄
pos	写入数据偏移量，未使用，可取 0 值

参数	描述
buffer	输出参数，指向定时器超时时间结构体的指针
size	超时时间结构体的大小
返回	——
超时时间结构体的 大小	成功
0	失败

使用示例如下所示：

```
rt_hwtimerval_t timeout_s; /* 用于保存定时器经过时间 */
/* 读取定时器经过时间 */
rt_device_read(hw_dev, 0, &timeout_s, sizeof(timeout_s));
```

15.2.7 关闭定时器设备

通过如下函数可以关闭定时器设备：

```
rt_err_t rt_device_close(rt_device_t dev);
```

参数	描述
dev	定时器设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

关闭设备接口和打开设备接口需配对使用，打开一次设备对应要关闭一次设备，这样设备才会被完全关闭，否则设备仍处于未关闭状态。

使用示例如下所示：

```
#define HW_TIMER_DEV_NAME "timer0" /* 定时器名称 */
rt_device_t hw_dev; /* 定时器设备句柄 */
/* 查找定时器设备 */
hw_dev = rt_device_find(HW_TIMER_DEV_NAME);
...
rt_device_close(hw_dev);
```

!!! note “注意事项” 可能出现定时误差。假设计数器最大值 0xFFFF，计数频率 1Mhz，定时时间 1 秒又 1 微秒。由于定时器一次最多只能计时到 65535us，对于 1000001us 的定时要求。可以 50000us 定时

20 次完成，此时将会出现计算误差 1us。

15.3 硬件定时器设备使用示例

硬件定时器设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 首先根据定时器设备名称“timer0”查找设备获取设备句柄。
2. 以读写方式打开设备“timer0”。
3. 设置定时器超时回调函数。
4. 设置定时器模式为周期性定时器，并设置超时时间为 5 秒，此时定时器启动。
5. 延时 3500ms 后读取定时器时间，读取到的值会以秒和微秒的形式显示。

```
/*
 * 程序清单：这是一个 hwtimer 设备使用例程
 * 例程导出了 hwtimer_sample 命令到控制终端
 * 命令调用格式：hwtimer_sample
 * 程序功能：硬件定时器超时回调函数周期性的打印当前tick值，2次tick值之差换算为时间等
 * 同于定时时间值。
 */

#include <rtthread.h>
#include <rtdevice.h>

#define HWTIMER_DEV_NAME    "timer0"      /* 定时器名称 */

/* 定时器超时回调函数 */
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback fucntion!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());

    return 0;
}

static int hwtimer_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    rt_hwtimerval_t timeout_s;      /* 定时器超时值 */
    rt_device_t hw_dev = RT_NULL;   /* 定时器设备句柄 */
    rt_hwtimer_mode_t mode;        /* 定时器模式 */

    /* 查找定时器设备 */
    hw_dev = rt_device_find(HWTIMER_DEV_NAME);
    if (hw_dev == RT_NULL)
    {
```

```
rt_kprintf("hwtimer sample run failed! can't find %s device!\n",
            HWTIMER_DEV_NAME);
return RT_ERROR;
}

/* 以读写方式打开设备 */
ret = rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
if (ret != RT_EOK)
{
    rt_kprintf("open %s device failed!\n", HWTIMER_DEV_NAME);
    return ret;
}

/* 设置超时回调函数 */
rt_device_set_rx_indicate(hw_dev, timeout_cb);

/* 设置模式为周期性定时器 */
mode = HWTIMER_MODE_PERIOD;
ret = rt_device_control(hw_dev, HWTIMER_CTRL_MODE_SET, &mode);
if (ret != RT_EOK)
{
    rt_kprintf("set mode failed! ret is :%d\n", ret);
    return ret;
}

/* 设置定时器超时值为5s并启动定时器 */
timeout_s.sec = 5;          /* 秒 */
timeout_s.usec = 0;          /* 微秒 */

if (rt_device_write(hw_dev, 0, &timeout_s, sizeof(timeout_s)) != sizeof(
    timeout_s))
{
    rt_kprintf("set timeout value failed\n");
    return RT_ERROR;
}

/* 延时3500ms */
rt_thread_mdelay(3500);

/* 读取定时器当前值 */
rt_device_read(hw_dev, 0, &timeout_s, sizeof(timeout_s));
rt_kprintf("Read: Sec = %d, Usec = %d\n", timeout_s.sec, timeout_s.usec);

return ret;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(hwtimer_sample, hwtimer sample);
```

第 16 章

I2C 总线设备

16.1 I2C 简介

I2C (Inter Integrated Circuit) 总线是 PHILIPS 公司开发的一种半双工、双向二线制同步串行总线。I2C 总线传输数据时只需两根信号线，一根是双向数据线 SDA (serial data)，另一根是双向时钟线 SCL (serial clock)。SPI 总线有两根线分别用于主从设备之间接收数据和发送数据，而 I2C 总线只使用一根线进行数据收发。

I2C 和 SPI 一样以主从的方式工作，不同于 SPI 一主多从的结构，它允许同时有多个主设备存在，每个连接到总线上的器件都有唯一的地址，主设备启动数据传输并产生时钟信号，从设备被主设备寻址，同一时刻只允许有一个主设备。如下图所示：

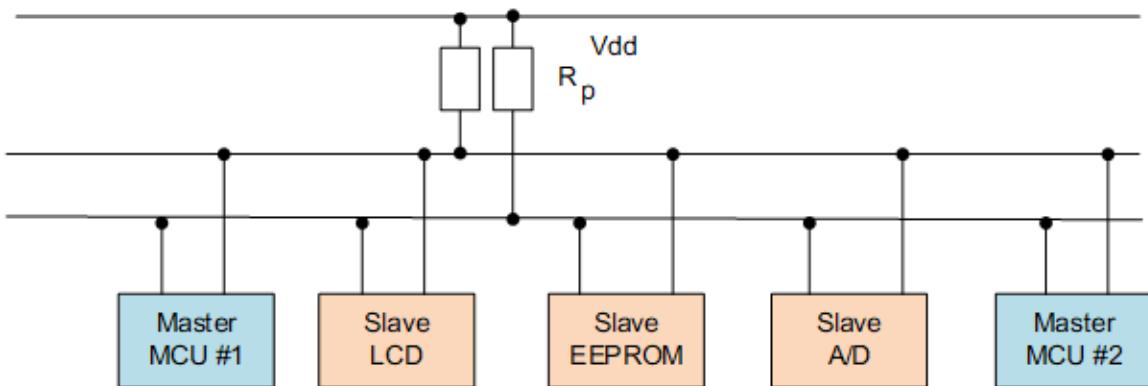


图 16.1: I2C 总线主从设备连接方式

如下图所示为 I2C 总线主要的数据传输格式：

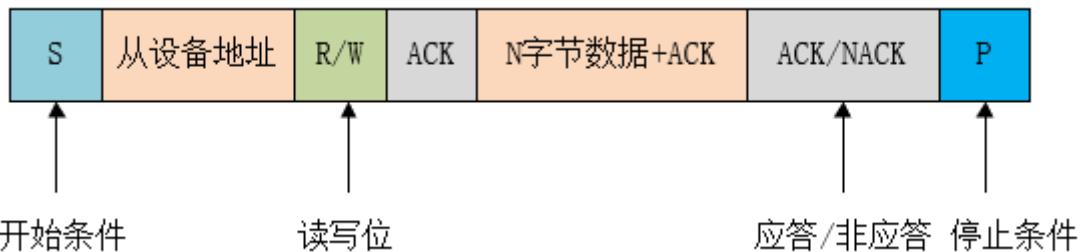


图 16.2: I2C 总线数据传输格式

当总线空闲时，SDA 和 SCL 都处于高电平状态，当主机要和某个从机通讯时，会先发送一个开始条件，然后发送从机地址和读写控制位，接下来传输数据（主机发送或者接收数据），数据传输结束时主机会发送停止条件。传输的每个字节为 8 位，高位在前，低位在后。数据传输过程中的不同名词详解如下所示：

- **开始条件：** SCL 为高电平时，主机将 SDA 拉低，表示数据传输即将开始。
- **从机地址：** 主机发送的第一个字节为从机地址，高 7 位为地址，最低位为 R/W 读写控制位，1 表示读操作，0 表示写操作。一般从机地址有 7 位地址模式和 10 位地址模式两种，如果是 10 位地址模式，第一个字节的头 7 位是 11110XX 的组合，其中最后两位（XX）是 10 位地址的两个最高位，第二个字节为 10 位从机地址的剩下 8 位，如下图所示：

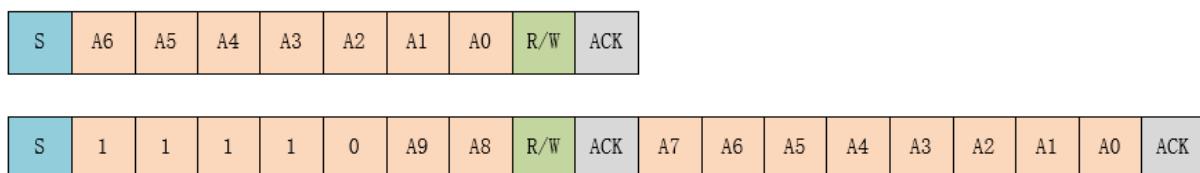


图 16.3: 7 位地址和 10 位地址格式

- **应答信号：** 每传输完成一个字节的数据，接收方就需要回复一个 ACK (acknowledge)。写数据时由从机发送 ACK，读数据时由主机发送 ACK。当主机读到最后一个字节数据时，可发送 NACK (Not acknowledge) 然后跟停止条件。
- **数据：** 从机地址发送完后可能会发送一些指令，依从机而定，然后开始传输数据，由主机或者从机发送，每个数据为 8 位，数据的字节数没有限制。
- **重复开始条件：** 在一次通信过程中，主机可能需要和不同的从机传输数据或者需要切换读写操作时，主机可以再发送一个开始条件。
- **停止条件：** 在 SDA 为低电平时，主机将 SCL 拉高并保持高电平，然后在将 SDA 拉高，表示传输结束。

16.2 访问 I2C 总线设备

一般情况下 MCU 的 I2C 器件都是作为主机和从机通讯，在 RT-Thread 中将 I2C 主机虚拟为 I2C 总线设备，I2C 从机通过 I2C 设备接口和 I2C 总线通讯，相关接口如下所示：

函数	描述
rt_device_find()	根据 I2C 总线设备名称查找设备获取设备句柄
rt_i2c_transfer()	传输数据

16.2.1 查找 I2C 总线设备

在使用 I2C 总线设备前需要根据 I2C 总线设备名称获取设备句柄，进而才可以操作 I2C 总线设备，查找设备函数如下所示，

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	I2C 总线设备名称
返回	—
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

一般情况下，注册到系统的 I2C 设备名称为 i2c0，i2c1 等，使用示例如下所示：

```
#define AHT10_I2C_BUS_NAME      "i2c1" /* 传感器连接的I2C总线设备名称 */
struct rt_i2c_bus_device *i2c_bus;        /* I2C总线设备句柄 */

/* 查找I2C总线设备，获取I2C总线设备句柄 */
i2c_bus = (struct rt_i2c_bus_device *)rt_device_find(name);
```

16.2.2 数据传输

获取到 I2C 总线设备句柄就可以使用 `rt_i2c_transfer()` 进行数据传输。函数原型如下所示：

```
rt_size_t rt_i2c_transfer(struct rt_i2c_bus_device *bus,
                           struct rt_i2c_msg          msgs[],
                           rt_uint32_t                 num);
```

参数	描述
bus	I2C 总线设备句柄
msgs[]	待传输的消息数组指针
num	消息数组的元素个数
返回	—
消息数组的元素个数	成功

参数	描述
错误码	失败

和 SPI 总线的自定义传输接口一样，I2C 总线的自定义传输接口传输的数据也是以一个消息为单位。参数 `msgs[]` 指向待传输的消息数组，用户可以自定义每条消息的内容，实现 I2C 总线所支持的 2 种不同的数据传输模式。如果主设备需要发送重复开始条件，则需要发送 2 个消息。

!!! note “注意事项” 此函数会调用 `rt_mutex_take()`, 不能在中断服务程序里面调用，会导致 assertion 报错。

I2C 消息数据结构原型如下：

```
struct rt_i2c_msg
{
    rt_uint16_t addr;      /* 从机地址 */
    rt_uint16_t flags;     /* 读、写标志等 */
    rt_uint16_t len;       /* 读写数据字节数 */
    rt_uint8_t *buf;       /* 读写数据缓冲区指针 */
}
```

从机地址 `addr`: 支持 7 位和 10 位二进制地址，需查看不同设备的数据手册。

!!! note “注意事项” RT-Thread I2C 设备接口使用的从机地址均不包含读写位，读写位控制需修改标志 `flags`。

标志 `flags` 可取值为以下宏定义，根据需要可以与其他宏使用位运算“|”组合起来使用。

#define RT_I2C_WR	0x0000	/* 写标志 */
#define RT_I2C_RD	(1u << 0)	/* 读标志 */
#define RT_I2C_ADDR_10BIT	(1u << 2)	/* 10 位地址模式 */
#define RT_I2C_NO_START	(1u << 4)	/* 无开始条件 */
#define RT_I2C_IGNORE_NACK	(1u << 5)	/* 忽视 NACK */
#define RT_I2C_NO_READ_ACK	(1u << 6)	/* 读的时候不发送 ACK */

使用示例如下所示：

```
#define AHT10_I2C_BUS_NAME      "i2c1"  /* 传感器连接的I2C总线设备名称 */
#define AHT10_ADDR                0x38    /* 从机地址 */
struct rt_i2c_bus_device *i2c_bus;        /* I2C总线设备句柄 */

/* 查找I2C总线设备，获取I2C总线设备句柄 */
i2c_bus = (struct rt_i2c_bus_device *)rt_device_find(name);

/* 读传感器寄存器数据 */
static rt_err_t read_regs(struct rt_i2c_bus_device *bus, rt_uint8_t len, rt_uint8_t_t
    *buf)
{
    struct rt_i2c_msg msgs;
```

```

msgs.addr = AHT10_ADDR;           /* 从机地址 */
msgs.flags = RT_I2C_RD;           /* 读标志 */
msgs.buf = buf;                  /* 读写数据缓冲区指针 */
msgs.len = len;                  /* 读写数据字节数 */

/* 调用I2C设备接口传输数据 */
if (rt_i2c_transfer(bus, &msgs, 1) == 1)
{
    return RT_EOK;
}
else
{
    return -RT_ERROR;
}
}

```

16.3 I2C 总线设备使用示例

I2C 设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

- 首先根据 I2C 设备名称查找 I2C 名称，获取设备句柄，然后初始化 aht10 传感器。
- 控制传感器的 2 的函数为写传感器寄存器 write_reg() 和读传感器寄存器 read_regs()，这两个函数分别调用了 rt_i2c_transfer() 传输数据。读取温湿度信息的函数 read_temp_humi() 则是调用这两个函数完成功能。

```

/*
 * 程序清单：这是一个 I2C 设备使用例程
 * 例程导出了 i2c_aht10_sample 命令到控制终端
 * 命令调用格式：i2c_aht10_sample i2c1
 * 命令解释：命令第二个参数是要使用的I2C总线设备名称，为空则使用默认的I2C总线设备
 * 程序功能：通过 I2C 设备读取温湿度传感器 aht10 的温湿度数据并打印
*/

#include <rtthread.h>
#include <rtdevice.h>

#define AHT10_I2C_BUS_NAME          "i2c1" /* 传感器连接的I2C总线设备名称 */
#define AHT10_ADDR                  0x38  /* 从机地址 */
#define AHT10_CALIBRATION_CMD       0xE1  /* 校准命令 */
#define AHT10_NORMAL_CMD             0xA8  /* 一般命令 */
#define AHT10_GET_DATA               0xAC  /* 获取数据命令 */

static struct rt_i2c_bus_device *i2c_bus = RT_NULL;      /* I2C总线设备句柄 */
static rt_bool_t initialized = RT_FALSE;                  /* 传感器初始化状态 */

/* 写传感器寄存器 */

```

```
static rt_err_t write_reg(struct rt_i2c_bus_device *bus, rt_uint8_t reg, rt_uint8_t_t
                          *data)
{
    rt_uint8_t buf[3];
    struct rt_i2c_msg msgs;

    buf[0] = reg; //cmd
    buf[1] = data[0];
    buf[2] = data[1];

    msgs.addr = AHT10_ADDR;
    msgs.flags = RT_I2C_WR;
    msgs.buf = buf;
    msgs.len = 3;

    /* 调用I2C设备接口传输数据 */
    if (rt_i2c_transfer(bus, &msgs, 1) == 1)
    {
        return RT_EOK;
    }
    else
    {
        return -RT_ERROR;
    }
}

/* 读传感器寄存器数据 */
static rt_err_t read_regs(struct rt_i2c_bus_device *bus, rt_uint8_t len, rt_uint8_t_t
                          *buf)
{
    struct rt_i2c_msg msgs;

    msgs.addr = AHT10_ADDR;
    msgs.flags = RT_I2C_RD;
    msgs.buf = buf;
    msgs.len = len;

    /* 调用I2C设备接口传输数据 */
    if (rt_i2c_transfer(bus, &msgs, 1) == 1)
    {
        return RT_EOK;
    }
    else
    {
        return -RT_ERROR;
    }
}

static void read_temp_humi(float *cur_temp, float *cur_humi)
```

```
{  
    rt_uint8_t temp[6];  
  
    write_reg(i2c_bus, AHT10_GET_DATA, 0); /* 发送命令 */  
    rt_thread_mdelay(400);  
    read_regs(i2c_bus, 6, temp); /* 获取传感器数据 */  
  
    /* 湿度数据转换 */  
    *cur_humi = (temp[1] << 12 | temp[2] << 4 | (temp[3] & 0xf0) >> 4) * 100.0 / (1  
     << 20);  
    /* 温度数据转换 */  
    *cur_temp = ((temp[3] & 0xf) << 16 | temp[4] << 8 | temp[5]) * 200.0 / (1 << 20)  
     - 50;  
}  
  
static void aht10_init(const char *name)  
{  
    rt_uint8_t temp[2] = {0, 0};  
  
    /* 查找I2C总线设备，获取I2C总线设备句柄 */  
    i2c_bus = (struct rt_i2c_bus_device *)rt_device_find(name);  
  
    if (i2c_bus == RT_NULL)  
    {  
        rt_kprintf("can't find %s device!\n", name);  
    }  
    else  
    {  
        write_reg(i2c_bus, AHT10_NORMAL_CMD, temp);  
        rt_thread_mdelay(400);  
  
        temp[0] = 0x08;  
        temp[1] = 0x00;  
        write_reg(i2c_bus, AHT10_CALIBRATION_CMD, temp);  
        rt_thread_mdelay(400);  
        initialized = RT_TRUE;  
    }  
}  
  
static void i2c_aht10_sample(int argc, char *argv[])  
{  
    float humidity, temperature;  
    char name[RT_NAME_MAX];  
  
    humidity = 0.0;  
    temperature = 0.0;  
  
    if (argc == 2)  
    {  
}
```

```
    rt_strncpy(name, argv[1], RT_NAME_MAX);
}
else
{
    rt_strncpy(name, AHT10_I2C_BUS_NAME, RT_NAME_MAX);
}

if (!initialized)
{
    /* 传感器初始化 */
    aht10_init(name);
}
if (initialized)
{
    /* 读取温湿度数据 */
    read_temp_humi(&temperature, &humidity);

    rt_kprintf("read aht10 sensor humidity : %d.%d %%\n", (int)humidity, (int)
               (humidity * 10) % 10);
    if( temperature >= 0 )
    {
        rt_kprintf("read aht10 sensor temperature: %d.%d°C\n", (int)temperature,
                   (int)(temperature * 10) % 10);
    }
    else
    {
        rt_kprintf("read aht10 sensor temperature: %d.%d°C\n", (int)temperature,
                   (int)(-temperature * 10) % 10);
    }
}
else
{
    rt_kprintf("initialize sensor failed!\n");
}
/*
 * 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(i2c_aht10_sample, i2c aht10 sample);
```

第 17 章

PWM 设备

17.1 PWM 简介

PWM(Pulse Width Modulation, 脉冲宽度调制) 是一种对模拟信号电平进行数字编码的方法，通过不同频率的脉冲使用方波的占空比用来对一个具体模拟信号的电平进行编码，使输出端得到一系列幅值相等的脉冲，用这些脉冲来代替所需要波形的设备。

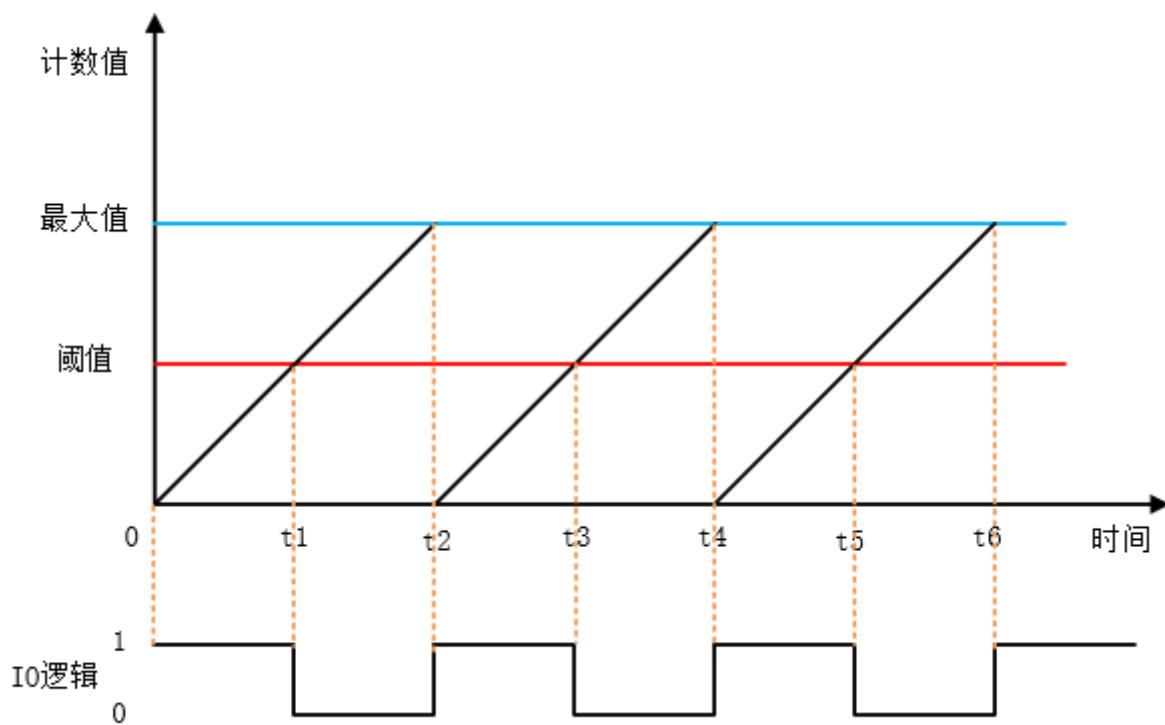


图 17.1: PWM 原理图

上图是一个简单的 PWM 原理示意图，假定定时器工作模式为向上计数，当计数值小于阈值时，则输出一种电平状态，比如高电平，当计数值大于阈值时则输出相反的电平状态，比如低电平。当计数值达到最大值时，计数器从 0 开始重新计数，又回到最初的电平状态。高电平持续时间（脉冲宽度）和周期时间的比值就是占空比，范围为 0~100%。上图高电平的持续时间刚好是周期时间的一半，所以占空比为 50%。

一个比较常用的 pwm 控制情景就是用来调节灯或者屏幕的亮度，根据占空比的不同，就可以完成亮度的调节。PWM 调节亮度并不是持续发光的，而是在不停地点亮、熄灭屏幕。当亮、灭交替够快时，肉眼就会认为一直在亮。在亮、灭的过程中，灭的状态持续时间越长，屏幕给肉眼的观感就是亮度越低。亮的时间越长，灭的时间就相应减少，屏幕就会变亮。

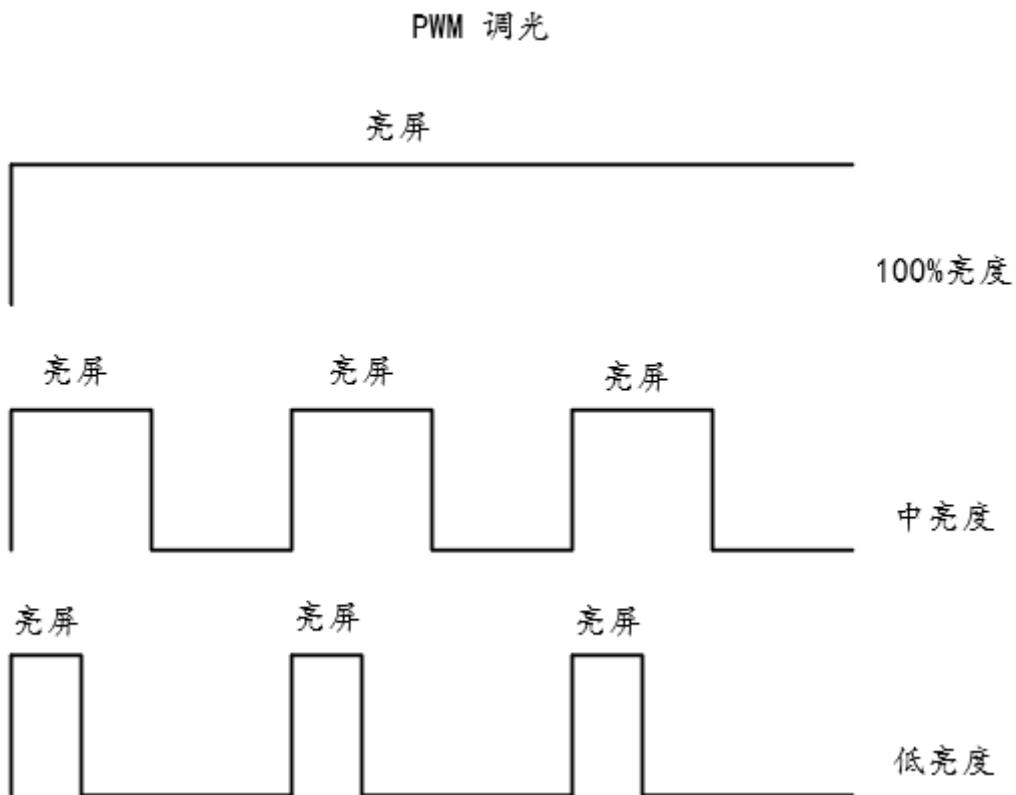


图 17.2: PWM 调节亮度

17.2 访问 PWM 设备

应用程序通过 RT-Thread 提供的 PWM 设备管理接口来访问 PWM 设备硬件，相关接口如下所示：

函数	描述
<code>rt_device_find()</code>	根据 PWM 设备名称查找设备获取设备句柄
<code>rt_pwm_set()</code>	设置 PWM 周期和脉冲宽度
<code>rt_pwm_enable()</code>	使能 PWM 设备
<code>rt_pwm_disable()</code>	关闭 PWM 设备

17.2.1 查找 PWM 设备

应用程序根据 PWM 设备名称获取设备句柄，进而可以操作 PWM 设备，查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	设备名称
返回	—
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

一般情况下，注册到系统的 PWM 设备名称为 pwm0, pwm1 等，使用示例如下所示：

```
#define PWM_DEV_NAME      "pwm3" /* PWM 设备名称 */
struct rt_device_pwm *pwm_dev; /* PWM 设备句柄 */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
```

17.2.2 设置 PWM 周期和脉冲宽度

通过如下函数设置 PWM 周期和占空比：

```
rt_err_t rt_pwm_set(struct rt_device_pwm *device,
                     int channel,
                     rt_uint32_t period,
                     rt_uint32_t pulse);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
period	PWM 周期时间 (单位纳秒 ns)
pulse	PWM 脉冲宽度时间 (单位纳秒 ns)
返回	—
RT_EOK	成功
-RT_EIO	device 为空
-RT_ENOSYS	设备操作方法为空
其他错误码	执行失败

PWM 的输出频率由周期时间 period 决定，例如周期时间为 0.5ms (毫秒)，则 period 值为 500000ns (纳秒)，输出频率为 2KHz，占空比为 pulse / period，pulse 值不能超过 period。

使用示例如下所示：

```
#define PWM_DEV_NAME      "pwm3" /* PWM设备名称 */
#define PWM_DEV_CHANNEL    4      /* PWM通道 */
struct rt_device_pwm *pwm_dev; /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000; /* 周期为0.5ms，单位为纳秒ns */
pulse = 0; /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
```

17.2.3 使能 PWM 设备

设置好 PWM 周期和脉冲宽度后就可以通过如下函数使能 PWM 设备：

```
rt_err_t rt_pwm_enable(struct rt_device_pwm *device, int channel);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
返回	—
RT_EOK	设备使能成功
-RT_ENOSYS	设备操作方法为空
其他错误码	设备使能失败

使用示例如下所示：

```
#define PWM_DEV_NAME      "pwm3" /* PWM设备名称 */
#define PWM_DEV_CHANNEL    4      /* PWM通道 */
struct rt_device_pwm *pwm_dev; /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000; /* 周期为0.5ms，单位为纳秒ns */
pulse = 0; /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
/* 使能设备 */
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);
```

17.2.4 关闭 PWM 设备通道

通过如下函数关闭 PWM 设备对应通道。

```
rt_err_t rt_pwm_disable(struct rt_device_pwm *device, int channel);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
返回	—
RT_EOK	设备关闭成功
-RT_EIO	设备句柄为空
其他错误码	设备关闭失败

使用示例如下所示：

```
#define PWM_DEV_NAME      "pwm3" /* PWM设备名称 */
#define PWM_DEV_CHANNEL    4      /* PWM通道 */
struct rt_device_pwm *pwm_dev; /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000; /* 周期为0.5ms，单位为纳秒ns */
pulse = 0;        /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
/* 使能设备 */
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);
/* 关闭设备通道 */
rt_pwm_disable(pwm_dev, PWM_DEV_CHANNEL);
```

17.3 FinSH 命令

设置 PWM 设备的某个通道的周期和占空比可使用命令 `pwm_set pwm1 1 500000 5000`，第一个参数为命令，第二个参数为 PWM 设备名称，第 3 个参数为 PWM 通道，第 4 个参数为周期（单位纳秒），第 5 个参数为脉冲宽度（单位纳秒）。

```
msh />pwm_set pwm1 1 500000 5000
msh />
```

使能 PWM 设备的某个通道可使用命令 `pwm_enable pwm1 1`，第一个参数为命令，第二个参数为 PWM 设备名称，第 3 个参数为 PWM 通道。

```
msh />pwm_enable pwm1 1
msh />
```

关闭 PWM 设备的某个通道可使用命令 `pwm_disable pwm1 1`, 第一个参数为命令, 第二个参数为 PWM 设备名称, 第 3 个参数为 PWM 通道。

```
msh />pwm_disable pwm1 1
msh />
```

17.4 PWM 设备使用示例

PWM 设备的具体使用方式可以参考如下示例代码, 示例代码的主要步骤如下:

1. 查找 PWM 设备获取设备句柄。
2. 设置 PWM 周期和脉冲宽度。
3. 使能 PWM 设备。
4. while 循环里每 50 毫秒修改一次脉冲宽度。
 - 将 PWM 通道对应引脚和 LED 对应引脚相连, 可以看到 LED 不停的由暗变到亮, 然后又从亮变到暗。

```
/*
 * 程序清单: 这是一个 PWM 设备使用例程
 * 例程导出了 pwm_led_sample 命令到控制终端
 * 命令调用格式: pwm_led_sample
 * 程序功能: 通过 PWM 设备控制 LED 灯的亮度, 可以看到LED不停的由暗变到亮, 然后又从亮
 * 变到暗。
 */

#include <rtthread.h>
#include <rtdevice.h>

#define PWM_DEV_NAME          "pwm3"    /* PWM设备名称 */
#define PWM_DEV_CHANNEL        4         /* PWM通道 */

struct rt_device_pwm *pwm_dev;        /* PWM设备句柄 */

static int pwm_led_sample(int argc, char *argv[])
{
    rt_uint32_t period, pulse, dir;

    period = 500000;    /* 周期为0.5ms, 单位为纳秒ns */
    dir = 1;            /* PWM脉冲宽度值的增减方向 */
    pulse = 0;          /* PWM脉冲宽度值, 单位为纳秒ns */
```

```
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
if (pwm_dev == RT_NULL)
{
    rt_kprintf("pwm sample run failed! can't find %s device!\n", PWM_DEV_NAME);
    return RT_ERROR;
}

/* 设置PWM周期和脉冲宽度默认值 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
/* 使能设备 */
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);

while (1)
{
    rt_thread_mdelay(50);
    if (dir)
    {
        pulse += 5000;          /* 从0值开始每次增加5000ns */
    }
    else
    {
        pulse -= 5000;          /* 从最大值开始每次减少5000ns */
    }
    if (pulse >= period)
    {
        dir = 0;
    }
    if (0 == pulse)
    {
        dir = 1;
    }

    /* 设置PWM周期和脉冲宽度 */
    rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
}
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pwm_led_sample, pwm sample);
```

第 18 章

RTC 设备

18.1 RTC 简介

RTC（Real-Time Clock）实时时钟可以提供精确的实时时间，它可用于产生年、月、日、时、分、秒等信息。目前实时时钟芯片大多采用精度较高的晶体振荡器作为时钟源。有些时钟芯片为了在主电源掉电时还可以工作，会外加电池供电，使时间信息一直保持有效。

RT-Thread 的 RTC 设备为操作系统的时间系统提供了基础服务。面对越来越多的 IoT 场景，RTC 已经成为产品的标配，甚至在诸如 SSL 的安全传输过程中，RTC 已经成为不可或缺的部分。

18.2 访问 RTC 设备

应用程序通过 RTC 设备管理接口来访问 RTC 硬件，相关接口如下所示：

函数	描述
set_date()	设置日期，年、月、日
set_time()	设置时间，时、分、秒
time()	获取当前时间

18.2.1 设置日期

通过如下函数设置 RTC 设备当前日期值：

```
rt_err_t set_date(rt_uint32_t year, rt_uint32_t month, rt_uint32_t day)
```

参数	描述
year	待设置生效的年份
month	待设置生效的月份

参数	描述
day	待设置生效的日
返回	——
RT_EOK	设置成功
-RT_ERROR	失败，没有找到 rtc 设备
其他错误码	失败

使用示例如下所示：

```
/* 设置日期为2018年12月3号 */
set_date(2018, 12, 3);
```

18.2.2 设置时间

通过如下函数设置 RTC 设备当前时间值：

```
rt_err_t set_time(rt_uint32_t hour, rt_uint32_t minute, rt_uint32_t second)
```

参数	描述
hour	待设置生效的时
minute	待设置生效的分
second	待设置生效的秒
返回	——
RT_EOK	设置成功
-RT_ERROR	失败，没有找到 rtc 设备
其他错误码	失败

使用示例如下所示：

```
/* 设置时间为11点15分50秒 */
set_time(11, 15, 50);
```

18.2.3 获取当前时间

使用到 C 标准库中的时间 API 获取时间：

```
time_t time(time_t *t)
```

参数	描述
t	时间数据指针
返回	——
当前时间值	

使用示例如下所示：

```
time_t now;      /* 保存获取的当前时间值 */
/* 获取时间 */
now = time(RT_NULL);
/* 打印输出时间信息 */
rt_kprintf("%s\n", ctime(&now));
```

!!! note “注意事项” 目前系统内只允许存在一个 RTC 设备，且名称为 "rtc"。

18.3 功能配置

18.3.1 启用 Soft RTC（软件模拟 RTC）

在 menuconfig 中可以启用使用软件模拟 RTC 的功能，这个模式非常适用于对时间精度要求不高，没有硬件 RTC 的产品。配置选项如下所示：

```
RT-Thread Components □
Device Drivers:
  [-] Using RTC device drivers          /* 使用 RTC 设备驱动 */
  [ ] Using software simulation RTC device /* 使用软件模拟 RTC */
```

18.3.2 启用 NTP 时间自动同步

如果 RT-Thread 已接入互联网，可启用 NTP 时间自动同步功能，定期同步本地时间。

首先在 menuconfig 中按照如下选项开启 NTP 功能：

```
RT-Thread online packages □
IoT - internet of things □
netutils: Networking utilities for RT-Thread:
  [*] Enable NTP(Network Time Protocol) client
```

开启 NTP 后 RTC 的自动同步功能将会自动开启，还可以设置同步周期和首次同步的延时时间：

```
RT-Thread Components □
Device Drivers:
```

```

-* Using RTC device drivers          /* 使用 RTC 设备驱动 */
[ ] Using software simulation RTC device /* 使用软件模拟 RTC */
[*] Using NTP auto sync RTC time    /* 使用 NTP 自动同步 RTC 时间 */
(30) NTP first sync delay time(second) for network connect /* 首次执行 NTP 时间同步的延时。延时的目的在于，给网络连接预留一定的时间，尽量提高第一次执行 NTP 时间同步时的成功率。默认时间为 30S; */
(3600) NTP auto sync period(second) /* NTP 自动同步周期，单位为秒，默认一小时（即 3600S）同步一次。 */

```

18.4 FinSH 命令

输入 `date` 即可查看当前时间，大致效果如下：

```
msh />date
Fri Feb 16 01:11:56 2018
msh />
```

同样使用 `date` 命令，在命令后面再依次输入 年 月 日 时 分 秒（中间空格隔开，24H 制），设置当前时间为 2018-02-16 01:15:30，大致效果如下：

```
msh />date 2018 02 16 01 15 30
msh />
```

18.5 RTC 设备使用示例

RTC 设备的具体使用方式可以参考如下示例代码，首先设置了年月日时分秒信息，然后延时 3 秒后获取当前时间信息。

```

/*
 * 程序清单：这是一个 RTC 设备使用例程
 * 例程导出了 rtc_sample 命令到控制终端
 * 命令调用格式：rtc_sample
 * 程序功能：设置RTC设备的日期和时间，延时一段时间后获取当前时间并打印显示。
*/
#include <rtthread.h>
#include <rtdevice.h>

static int rtc_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    time_t now;

```

```
/* 设置日期 */
ret = set_date(2018, 12, 3);
if (ret != RT_EOK)
{
    rt_kprintf("set RTC date failed\n");
    return ret;
}

/* 设置时间 */
ret = set_time(11, 15, 50);
if (ret != RT_EOK)
{
    rt_kprintf("set RTC time failed\n");
    return ret;
}

/* 延时3秒 */
rt_thread_mdelay(3000);

/* 获取时间 */
now = time(RT_NULL);
rt_kprintf("%s\n", ctime(&now));

return ret;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(rtc_sample, rtc sample);
```

第 19 章

SPI 设备

19.1 SPI 简介

SPI (Serial Peripheral Interface, 串行外设接口) 是一种高速、全双工、同步通信总线，常用于短距离通讯，主要应用于 EEPROM、FLASH、实时时钟、AD 转换器、还有数字信号处理器和数字信号解码器之间。SPI 一般使用 4 根线通信，如下图所示：

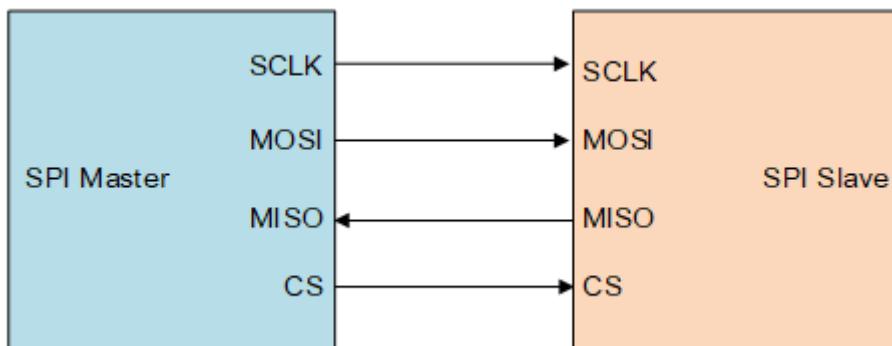


图 19.1: SPI 主设备和从设备的连接方式

- MOSI – 主机输出 / 从机输入数据线 (SPI Bus Master Output/Slave Input)。
- MISO – 主机输入 / 从机输出数据线 (SPI Bus Master Input/Slave Output)。
- SCLK – 串行时钟线 (Serial Clock)，主设备输出时钟信号至从设备。
- CS – 从设备选择线 (Chip select)。也叫 SS、CSB、CSN、EN 等，主设备输出片选信号至从设备。

SPI 以主从方式工作，通常有一个主设备和一个或多个从设备。通信由主设备发起，主设备通过 CS 选择要通信的从设备，然后通过 SCLK 给从设备提供时钟信号，数据通过 MOSI 输出给从设备，同时通过 MISO 接收从设备发送的数据。

如下图所示芯片有 2 个 SPI 控制器，SPI 控制器对应 SPI 主设备，每个 SPI 控制器可以连接多个 SPI 从设备。挂载在同一个 SPI 控制器上的从设备共享 3 个信号引脚：SCK、MISO、MOSI，但每个从设备的 CS 引脚是独立的。

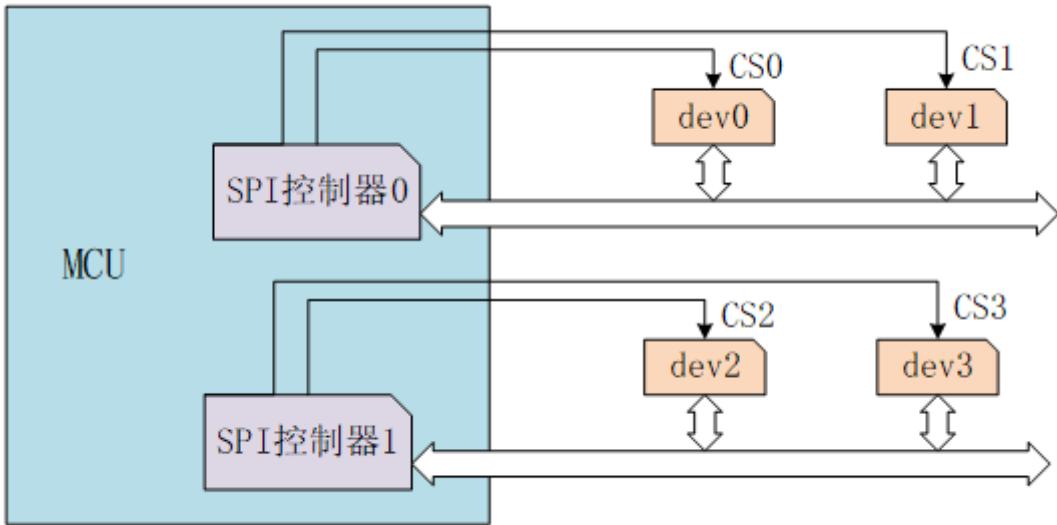


图 19.2: 一个 SPI 主设备与多个从设备连接

主设备通过控制 CS 引脚对从设备进行片选，一般为低电平有效。任何时刻，一个 SPI 主设备上只有一个 CS 引脚处于有效状态，与该有效 CS 引脚连接的从设备此时可以与主设备通信。

从设备的时钟由主设备通过 SCLK 提供，MOSI、MISO 则基于此脉冲完成数据传输。SPI 的工作时序模式由 CPOL（Clock Polarity，时钟极性）和 CPHA（Clock Phase，时钟相位）之间的相位关系决定，CPOL 表示时钟信号的初始电平的状态，CPOL 为 0 表示时钟信号初始状态为低电平，为 1 表示时钟信号的初始电平是高电平。CPHA 表示在哪个时钟沿采样数据，CPHA 为 0 表示在首个时钟变化沿采样数据，而 CPHA 为 1 则表示在第二个时钟变化沿采样数据。根据 CPOL 和 CPHA 的不同组合共有 4 种工作时序模式：CPOL=0, CPHA=0、CPOL=0, CPHA=1、CPOL=1, CPHA=0、CPOL=1, CPHA=1。如下图所示：

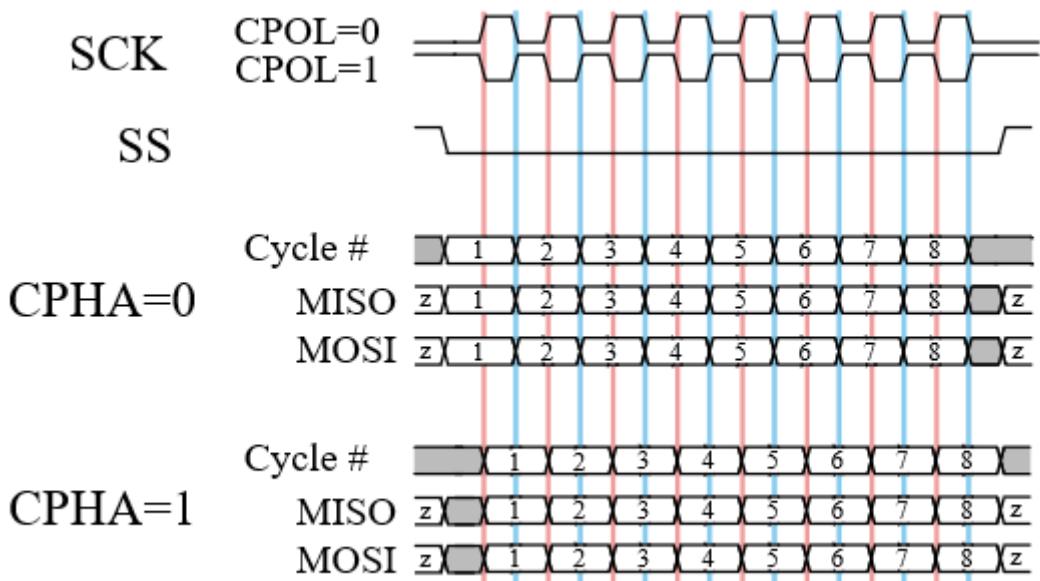


图 19.3: SPI 4 种工作模式时序图

QSPI: QSPI 是 Queued SPI 的简写，是 Motorola 公司推出的 SPI 接口的扩展，比 SPI 应用更加广泛。在 SPI 协议的基础上，Motorola 公司对其功能进行了增强，增加了队列传输机制，推出了队列串行外围接

口协议（即 QSPI 协议）。使用该接口，用户可以一次性传输包含多达 16 个 8 位或 16 位数据的传输队列。一旦传输启动，都不需要 CPU 干预，极大的提高了传输效率。与 SPI 相比，QSPI 的最大结构特点是以 80 字节的 RAM 代替了 SPI 的发送和接收数据寄存器。

Dual SPI Flash: 对于 SPI Flash 而言全双工并不常用，可以发送一个命令字节进入 Dual 模式，让它工作在半双工模式，用以加倍数据传输。这样 MOSI 变成 SIO0 (serial io 0)，MISO 变成 SIO1 (serial io 1)，这样一个时钟周期内就能传输 2 个 bit 数据，加倍了数据传输。

Quad SPI Flash: 与 Dual SPI 类似，Quad SPI Flash 增加了两根 I/O 线 (SIO2,SIO3)，目的是一个时钟内传输 4 个 bit 数据。

所以对于 SPI Flash，有标准 SPI Flash，Dual SPI Flash，Quad SPI Flash 三种类型。在相同时钟下，线数越多传输速率越高。

19.2 挂载 SPI 设备

SPI 驱动会注册 SPI 总线，SPI 设备需要挂载到已经注册好的 SPI 总线上。

```
rt_err_t rt_spi_bus_attach_device(struct rt_spi_device *device,
                                    const char          *name,
                                    const char          *bus_name,
                                    void                *user_data)
```

参数	描述
device	SPI 设备句柄
name	SPI 设备名称
bus_name	SPI 总线名称
user_data	用户数据指针
返回	—
RT_EOK	成功
其他错误码	失败

此函数用于挂载一个 SPI 设备到指定的 SPI 总线，并向内核注册 SPI 设备，并将 user_data 保存到 SPI 设备的控制块里。

一般 SPI 总线命名原则为 spix，SPI 设备命名原则为 spify，如 spi10 表示挂载在 spi1 总线上的 0 号设备。user_data 一般为 SPI 设备的 CS 引脚指针，进行数据传输时 SPI 控制器会操作此引脚进行片选。

若使用 rt-thread/bsp/stm32 目录下的 BSP 则可以使用下面的函数挂载 SPI 设备到总线：

```
rt_err_t rt_hw_spi_device_attach(const char *bus_name, const char *device_name,
                                 GPIO_TypeDef* cs_gpiox, uint16_t cs_gpio_pin);
```

下面的示例代码挂载 SPI FLASH W25Q128 到 SPI 总线：

```

static int rt_hw_spi_flash_init(void)
{
    __HAL_RCC_GPIOB_CLK_ENABLE();
    rt_hw_spi_device_attach("spi1", "spi10", GPIOB, GPIO_PIN_14);

    if (RT_NULL == rt_sfud_flash_probe("W25Q128", "spi10"))
    {
        return -RT_ERROR;
    };

    return RT_EOK;
}

/* 导出到自动初始化 */
INIT_COMPONENT_EXPORT(rt_hw_spi_flash_init);

```

19.3 配置 SPI 设备

挂载 SPI 设备到 SPI 总线后需要配置 SPI 设备的传输参数。

```
rt_err_t rt_spi_configure(struct rt_spi_device *device,
                           struct rt_spi_configuration *cfg)
```

参数	描述
device	SPI 设备句柄
cfg	SPI 配置参数指针
返回	—
RT_EOK	成功

此函数会保存 cfg 指向的配置参数到 SPI 设备 device 的控制块里，当传输数据时会使用此配置参数。

struct rt_spi_configuration 原型如下：

```
struct rt_spi_configuration
{
    rt_uint8_t mode;          /* 模式 */
    rt_uint8_t data_width;    /* 数据宽度，可取8位、16位、32位 */
    rt_uint16_t reserved;    /* 保留 */
    rt_uint32_t max_hz;      /* 最大频率 */
};
```

模式： 包含 MSB/LSB、主从模式、时序模式等，可取宏组合如下：

```
/* 设置数据传输顺序是MSB位在前还是LSB位在前 */
#define RT_SPI_LSB      (0<<2)           /* bit[2]: 0-LSB */
#define RT_SPI_MSB      (1<<2)           /* bit[2]: 1-MSB */
```

```

/* 设置 SPI 的主从模式 */
#define RT_SPI_MASTER    (0<<3)                      /* SPI master device */
#define RT_SPI_SLAVE     (1<<3)                      /* SPI slave device */

/* 设置时钟极性和时钟相位 */
#define RT_SPI_MODE_0    (0 | 0)                         /* CPOL = 0, CPHA = 0 */
#define RT_SPI_MODE_1    (0 | RT_SPI_CPHA)                /* CPOL = 0, CPHA = 1 */
#define RT_SPI_MODE_2    (RT_SPI_CPOL | 0)                /* CPOL = 1, CPHA = 0 */
#define RT_SPI_MODE_3    (RT_SPI_CPOL | RT_SPI_CPHA)      /* CPOL = 1, CPHA = 1 */

#define RT_SPI_CS_HIGH   (1<<4)                        /* Chipselect active high */
#define RT_SPI_NO_CS     (1<<5)                        /* No chipselect */
#define RT_SPI_3WIRE     (1<<6)                        /* SI/SO pin shared */
#define RT_SPI_READY     (1<<7)                        /* Slave pulls low to pause */

```

数据宽度：根据 SPI 主设备及 SPI 从设备可发送及接收的数据宽度格式设置为 8 位、16 位或者 32 位。

最大频率：设置数据传输的波特率，同样根据 SPI 主设备及 SPI 从设备工作的波特率范围设置。

配置示例如下所示：

```

struct rt_spi_configuration cfg;
cfg.data_width = 8;
cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
cfg.max_hz = 20 * 1000 *1000;                                /* 20M */

rt_spi_configure(spi_dev, &cfg);

```

19.4 配置 QSPI 设备

配置 QSPI 设备的传输参数可使用如下函数：

```

rt_err_t rt_qspi_configure(struct rt_qspi_device *device, struct
                           rt_qspi_configuration *cfg);

```

参数	描述
device	QSPI 设备句柄
cfg	QSPI 配置参数指针
返回	—
RT_EOK	成功

此函数会保存 cfg 指向的配置参数到 QSPI 设备 device 的控制块里，当传输数据时会使用此配置参数。

struct rt_qspi_configuration 原型如下：

```

struct rt_qspi_configuration
{
    struct rt_spi_configuration parent;          /* 继承自 SPI 设备配置参数 */
    rt_uint32_t medium_size;                    /* 介质大小 */
    rt_uint8_t ddr_mode;                         /* 双倍速率模式 */
    rt_uint8_t qspi_dl_width ;                   /* QSPI 总线位宽, 单线模式 1 位、双线模式
                                                2 位, 4 线模式 4 位 */
};

```

19.5 访问 SPI 设备

一般情况下 MCU 的 SPI 器件都是作为主机和从机通讯，在 RT-Thread 中将 SPI 主机虚拟为 SPI 总线设备，应用程序使用 SPI 设备管理接口来访问 SPI 从机器件，主要接口如下所示：

函数	描述
<code>rt_device_find()</code>	根据 SPI 设备名称查找设备获取设备句柄
<code>rt_spi_transfer_message()</code>	自定义传输数据
<code>rt_spi_transfer()</code>	传输一次数据
<code>rt_spi_send()</code>	发送一次数据
<code>rt_spi_recv()</code>	接受一次数据
<code>rt_spi_send_then_send()</code>	连续两次发送
<code>rt_spi_send_then_recv()</code>	先发送后接收

!!! note “注意事项” SPI 数据传输相关接口会调用 `rt_mutex_take()`, 此函数不能在中断服务程序里面调用，会导致 assertion 报错。

19.5.1 查找 SPI 设备

在使用 SPI 设备前需要根据 SPI 设备名称获取设备句柄，进而才可以操作 SPI 设备，查找设备函数如下所示，

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
<code>name</code>	设备名称
返回	—
设备句柄	查找到对应设备将返回相应的设备句柄
<code>RT_NULL</code>	没有找到相应的设备对象

一般情况下，注册到系统的 SPI 设备名称为 spi10, qspi10 等，使用示例如下所示：

```
#define W25Q_SPI_DEVICE_NAME      "qspi10"    /* SPI 设备名称 */
struct rt_spi_device *spi_dev_w25q;        /* SPI 设备句柄 */

/* 查找 spi 设备获取设备句柄 */
spi_dev_w25q = (struct rt_spi_device *)rt_device_find(W25Q_SPI_DEVICE_NAME);
```

19.5.2 自定义传输数据

获取到 SPI 设备句柄就可以使用 SPI 设备管理接口访问 SPI 设备器件，进行数据收发。可以通过如下函数传输消息：

```
struct rt_spi_message *rt_spi_transfer_message(struct rt_spi_device *device, struct
                                              rt_spi_message *message);
```

参数	描述
device	SPI 设备句柄
message	消息指针
返回	—
RT_NULL	成功发送
非空指针	发送失败，返回指向剩余未发送的 message 的指针

此函数可以传输一连串消息，用户可以自定义每个待传输的 message 结构体各参数的数值，从而可以很方便的控制数据传输方式。struct rt_spi_message 原型如下：

```
struct rt_spi_message
{
    const void *send_buf;           /* 发送缓冲区指针 */
    void *recv_buf;                /* 接收缓冲区指针 */
    rt_size_t length;              /* 发送 / 接收 数据字节数 */
    struct rt_spi_message *next;    /* 指向继续发送的下一条消息的指针 */
    unsigned cs_take : 1;           /* 片选选中 */
    unsigned cs_release : 1;         /* 释放片选 */
};
```

sendbuf 为发送缓冲区指针，其值为 RT_NULL 时，表示本次传输为只接收状态，不需要发送数据。

recvbuf 为接收缓冲区指针，其值为 RT_NULL 时，表示本次传输为只发送状态，不需要保存接收到的数据，所以收到的数据直接丢弃。

length 的单位为 word，即数据长度为 8 位时，每个 length 占用 1 个字节；当数据长度为 16 位时，每个 length 占用 2 个字节。

参数 next 是指向继续发送的下一条消息的指针，若只发送一条消息，则此指针值为 RT_NULL。多个待传输的消息通过 next 指针以单向链表的形式连接在一起。

`cs_take` 值为 1 时，表示在传输数据前，设置对应的 CS 为有效状态。`cs_release` 值为 1 时，表示在数据传输结束后，释放对应的 CS。

!!! note “注意事项” * 当 `send_buf` 或 `recv_buf` 不为空时，两者的可用空间都不得小于 `length`。* 若使用此函数传输消息，传输的第一条消息 `cs_take` 需置为 1，设置片选为有效，最后一条消息的 `cs_release` 需置 1，释放片选。

使用示例如下所示：

```
#define W25Q_SPI_DEVICE_NAME      "qspi10"    /* SPI 设备名称 */
struct rt_spi_device *spi_dev_w25q;        /* SPI 设备句柄 */
struct rt_spi_message msg1, msg2;
rt_uint8_t w25x_read_id = 0x90;           /* 命令 */
rt_uint8_t id[5] = {0};

/* 查找 spi 设备获取设备句柄 */
spi_dev_w25q = (struct rt_spi_device *)rt_device_find(W25Q_SPI_DEVICE_NAME);
/* 发送命令读取ID */
struct rt_spi_message msg1, msg2;

msg1.send_buf    = &w25x_read_id;
msg1.recv_buf    = RT_NULL;
msg1.length     = 1;
msg1.cs_take    = 1;
msg1.cs_release  = 0;
msg1.next       = &msg2;

msg2.send_buf    = RT_NULL;
msg2.recv_buf    = id;
msg2.length     = 5;
msg2.cs_take    = 0;
msg2.cs_release  = 1;
msg2.next       = RT_NULL;

rt_spi_transfer_message(spi_dev_w25q, &msg1);
rt_kprintf("use rt_spi_transfer_message() read w25q ID is:%x%x\n", id[3], id[4]);
```

19.5.3 传输一次数据

如果只传输一次数据可以通过如下函数：

```
rt_size_t rt_spi_transfer(struct rt_spi_device *device,
                           const void          *send_buf,
                           void                *recv_buf,
                           rt_size_t            length);
```

参数	描述
device	SPI 设备句柄

参数	描述
send_buf	发送数据缓冲区指针
recv_buf	接收数据缓冲区指针
length	发送/接收数据字节数
返回	—
0	传输失败
非 0 值	成功传输的字节数

此函数等同于调用 `rt_spi_transfer_message()` 传输一条消息，开始发送数据时片选选中，函数返回时释放片选，`message` 参数配置如下：

```
struct rt_spi_message msg;

msg.send_buf    = send_buf;
msg.recv_buf    = recv_buf;
msg.length     = length;
msg.cs_take    = 1;
msg.cs_release = 1;
msg.next       = RT_NULL;
```

19.5.4 发送一次数据

如果只发送一次数据，而忽略接收到的数据可以通过如下函数：

```
rt_size_t rt_spi_send(struct rt_spi_device *device,
                      const void           *send_buf,
                      rt_size_t             length)
```

参数	描述
device	SPI 设备句柄
send_buf	发送数据缓冲区指针
length	发送数据字节数
返回	—
0	发送失败
非 0 值	成功发送的字节数

调用此函数发送 `send_buf` 指向的缓冲区的数据，忽略接收到的数据，此函数是对 `rt_spi_transfer()` 函数的封装。

此函数等同于调用 `rt_spi_transfer_message()` 传输一条消息，开始发送数据时片选选中，函数返

回时释放片选, message 参数配置如下:

```
struct rt_spi_message msg;

msg.send_buf = send_buf;
msg.recv_buf = RT_NULL;
msg.length = length;
msg.cs_take = 1;
msg.cs_release = 1;
msg.next = RT_NULL;
```

19.5.5 接收一次数据

如果只接收一次数据可以通过如下函数:

```
rt_size_t rt_spi_recv(struct rt_spi_device *device,
                      void *recv_buf,
                      rt_size_t length);
```

参数	描述
device	SPI 设备句柄
recv_buf	接收数据缓冲区指针
length	接收数据字节数
返回	—
0	接收失败
非 0 值	成功接收的字节数

调用此函数接收数据并保存到 recv_buf 指向的缓冲区。此函数是对 rt_spi_transfer() 函数的封装。SPI 总线协议规定只能由主设备产生时钟, 因此在接收数据时, 主设备会发送数据 0xFF。

此函数等同于调用 rt_spi_transfer_message() 传输一条消息, 开始接收数据时片选选中, 函数返回时释放片选, message 参数配置如下:

```
struct rt_spi_message msg;

msg.send_buf = RT_NULL;
msg.recv_buf = recv_buf;
msg.length = length;
msg.cs_take = 1;
msg.cs_release = 1;
msg.next = RT_NULL;
```

19.5.6 连续两次发送数据

如果需要先后连续发送 2 个缓冲区的数据，并且中间片选不释放，可以调用如下函数：

```
rt_err_t rt_spi_send_then_send(struct rt_spi_device *device,
                                const void          *send_buf1,
                                rt_size_t            send_length1,
                                const void          *send_buf2,
                                rt_size_t            send_length2);
```

参数	描述
device	SPI 设备句柄
send_buf1	发送数据缓冲区 1 指针
send_length1	发送数据缓冲区 1 数据字节数
send_buf2	发送数据缓冲区 2 指针
send_length2	发送数据缓冲区 2 数据字节数
返回	—
RT_EOK	发送成功
-RT_EIO	发送失败

此函数可以连续发送 2 个缓冲区的数据，忽略接收到的数据，发送 `send_buf1` 时片选选中，发送完 `send_buf2` 后释放片选。

本函数适合向 SPI 设备中写入一块数据，第一次先发送命令和地址等数据，第二次再发送指定长度的数据。之所以分两次发送而不是合并成一个数据块发送，或调用两次 `rt_spi_send()`，是因为在大部分的数据写操作中，都需要先发命令和地址，长度一般只有几个字节。如果与后面的数据合并在一起发送，将需要进行内存空间申请和大量的数据搬运。而如果调用两次 `rt_spi_send()`，那么在发送完命令和地址后，片选会被释放，大部分 SPI 设备都依靠设置片选一次有效为命令的起始，所以片选在发送完命令或地址数据后被释放，则此次操作被丢弃。

此函数等同于调用 `rt_spi_transfer_message()` 传输 2 条消息，`message` 参数配置如下：

```
struct rt_spi_message msg1,msg2;

msg1.send_buf    = send_buf1;
msg1.recv_buf    = RT_NULL;
msg1.length     = send_length1;
msg1.cs_take    = 1;
msg1.cs_release = 0;
msg1.next       = &msg2;

msg2.send_buf    = send_buf2;
msg2.recv_buf    = RT_NULL;
msg2.length     = send_length2;
msg2.cs_take    = 0;
```

```
msg2.cs_release = 1;
msg2.next       = RT_NULL;
```

19.5.7 先发送后接收数据

如果需要向从设备先发送数据，然后接收从设备发送的数据，并且中间片选不释放，可以调用如下函数：

```
rt_err_t rt_spi_send_then_recv(struct rt_spi_device *device,
                               const void          *send_buf,
                               rt_size_t            send_length,
                               void                *recv_buf,
                               rt_size_t            recv_length);
```

参数	描述
device	SPI 从设备句柄
send_buf	发送数据缓冲区指针
send_length	发送数据缓冲区数据字节数
recv_buf	接收数据缓冲区指针
recv_length	接收数据字节数
返回	—
RT_EOK	成功
-RT_EIO	失败

此函数发送第一条数据 `send_buf` 时开始片选，此时忽略接收到的数据，然后发送第二条数据，此时主设备会发送数据 `0xFF`，接收到的数据保存在 `recv_buf` 里，函数返回时释放片选。

本函数适合从 SPI 从设备中读取一块数据，第一次会先发送一些命令和地址数据，然后再接收指定长度的数据。

此函数等同于调用 `rt_spi_transfer_message()` 传输 2 条消息，`message` 参数配置如下：

```
struct rt_spi_message msg1,msg2;

msg1.send_buf    = send_buf;
msg1.recv_buf    = RT_NULL;
msg1.length     = send_length;
msg1.cs_take    = 1;
msg1.cs_release = 0;
msg1.next       = &msg2;

msg2.send_buf    = RT_NULL;
msg2.recv_buf    = recv_buf;
msg2.length     = recv_length;
```

```
msg2.cs_take    = 0;
msg2.cs_release = 1;
msg2.next       = RT_NULL;
```

SPI 设备管理模块还提供 `rt_spi_sendrecv8()` 和 `rt_spi_sendrecv16()` 函数，这两个函数都是对此函数的封装，`rt_spi_sendrecv8()` 发送一个字节数据同时收到一个字节数据，`rt_spi_sendrecv16()` 发送 2 个字节数据同时收到 2 个字节数据。

19.6 访问 QSPI 设备

QSPI 的数据传输接口如下所示：

函数	描述
<code>rt_qspi_transfer_message()</code>	传输数据
<code>rt_qspi_send_then_recv()</code>	先发送后接收
<code>rt_qspi_send()</code>	发送一次数据

!!! note “注意事项” QSPI 数据传输相关接口会调用 `rt_mutex_take()`，此函数不能在中断服务程序里面调用，会导致 `assertion` 报错。

19.6.1 传输数据

可以通过如下函数传输消息：

```
rt_size_t rt_qspi_transfer_message(struct rt_qspi_device *device, struct
                                     rt_qspi_message *message);
```

参数	描述
<code>device</code>	QSPI 设备句柄
<code>message</code>	消息指针
返回	—
实际传输的消息大小	—

消息结构体 `struct rt_qspi_message` 原型如下：

```
struct rt_qspi_message
{
    struct rt_spi_message parent; /* 继承自 struct rt_spi_message */

    struct
```

```

{
    rt_uint8_t content;           /* 指令内容 */
    rt_uint8_t qspi_lines;        /* 指令模式, 单线模式 1 位、双线模式 2 位, 4 线
                                  模式 4 位 */
} instruction;                  /* 指令阶段 */

struct
{
    rt_uint32_t content;          /* 地址/交替字节 内容 */
    rt_uint8_t size;              /* 地址/交替字节 长度 */
    rt_uint8_t qspi_lines;        /* 地址/交替字节 模式, 单线模式 1 位、双线模式 2
                                  位, 4 线模式 4 位 */
} address, alternate_bytes;     /* 地址/交替字节 阶段 */

rt_uint32_t dummy_cycles;       /* 空指令周期阶段 */
rt_uint8_t qspi_data_lines;     /* QSPI 总线位宽 */
};


```

19.6.2 接收数据

可以调用如下函数:

```
rt_err_t rt_qspi_send_then_recv(struct rt_qspi_device *device,
                                const void *send_buf,
                                rt_size_t send_length,
                                void *recv_buf,
                                rt_size_t recv_length);
```

参数	描述
device	QSPI 设备句柄
send_buf	发送数据缓冲区指针
send_length	发送数据字节数
recv_buf	接收数据缓冲区指针
recv_length	接收数据字节数
返回	—
RT_EOK	成功
其他错误码	失败

send_buf 参数包含了将要发送的命令序列。

19.6.3 发送数据

```
rt_err_t rt_qspi_send(struct rt_qspi_device *device, const void *send_buf, rt_size_t length)
```

参数	描述
device	QSPI 设备句柄
send_buf	发送数据缓冲区指针
length	发送数据字节数
返回	—
RT_EOK	成功
其他错误码	失败

send_buf 参数包含了将要发送的命令序列和数据。

19.7 特殊使用场景

在一些特殊的使用场景，某个设备希望独占总线一段时间，且期间要保持片选一直有效，期间数据传输可能是间断的，则可以按照如所示步骤使用相关接口。传输数据函数必须使用 `rt_spi_transfer_message()`，并且此函数每个待传输消息的片选控制域 `cs_take` 和 `cs_release` 都要设置为 0 值，因为片选已经使用了其他接口控制，不需要在数据传输的时候控制。

19.7.1 获取总线

在多线程的情况下，同一个 SPI 总线可能会在不同的线程中使用，为了防止 SPI 总线正在传输的数据丢失，从设备在开始传输数据前需要先获取 SPI 总线的使用权，获取成功才能够使用总线传输数据，可使用如下函数获取 SPI 总线：

```
rt_err_t rt_spi_take_bus(struct rt_spi_device *device);
```

参数	描述
device	SPI 设备句柄
返回	—
RT_EOK	成功
错误码	失败

19.7.2 选中片选

从设备获取总线的使用权后，需要设置自己对应的片选信号为有效，可使用如下函数选中片选：

```
rt_err_t rt_spi_take(struct rt_spi_device *device);
```

参数	描述
device	SPI 设备句柄
返回	—
0	成功
错误码	失败

19.7.3 增加一条消息

使用 `rt_spi_transfer_message()` 传输消息时，所有待传输的消息都是以单向链表的形式连接起来的，可使用如下函数往消息链表里增加一条新的待传输消息：

```
void rt_spi_message_append(struct rt_spi_message *list,
                           struct rt_spi_message *message);
```

参数	描述
list	待传输的消息链表节点
message	新增消息指针

19.7.4 释放片选

从设备数据传输完成后，需要释放片选，可使用如下函数释放片选：

```
rt_err_t rt_spi_release(struct rt_spi_device *device);
```

参数	描述
device	SPI 设备句柄
返回	—
0	成功
错误码	失败

19.7.5 释放总线

从设备不在使用 SPI 总线传输数据，必须尽快释放总线，这样其他从设备才能使用 SPI 总线传输数据，可使用如下函数释放总线：

```
rt_err_t rt_spi_release_bus(struct rt_spi_device *device);
```

参数	描述
device	SPI 设备句柄
返回	—
RT_EOK	成功

19.8 SPI 设备使用示例

SPI 设备的具体使用方式可以参考如下的示例代码，示例代码首先查找 SPI 设备获取设备句柄，然后使用 `rt_spi_transfer_message()` 发送命令读取 ID 信息。

```
/*
 * 程序清单：这是一个 SPI 设备使用例程
 * 例程导出了 spi_w25q_sample 命令到控制终端
 * 命令调用格式：spi_w25q_sample spi10
 * 命令解释：命令第二个参数是要使用的SPI设备名称，为空则使用默认的SPI设备
 * 程序功能：通过SPI设备读取 w25q 的 ID 数据
 */

#include <rtthread.h>
#include <rtdevice.h>

#define W25Q_SPI_DEVICE_NAME      "qspi10"

static void spi_w25q_sample(int argc, char *argv[])
{
    struct rt_spi_device *spi_dev_w25q;
    char name[RT_NAME_MAX];
    rt_uint8_t w25x_read_id = 0x90;
    rt_uint8_t id[5] = {0};

    if (argc == 2)
    {
        rt_strncpy(name, argv[1], RT_NAME_MAX);
    }
    else
    {
        rt_strncpy(name, W25Q_SPI_DEVICE_NAME, RT_NAME_MAX);
    }

    /* 查找 spi 设备获取设备句柄 */
    spi_dev_w25q = (struct rt_spi_device *)rt_device_find(name);
    if (!spi_dev_w25q)
    {
```

```
    rt_kprintf("spi sample run failed! can't find %s device!\n", name);
}
else
{
    /* 方式1：使用 rt_spi_send_then_recv()发送命令读取ID */
    rt_spi_send_then_recv(spi_dev_w25q, &w25x_read_id, 1, id, 5);
    rt_kprintf("use rt_spi_send_then_recv() read w25q ID is:%x%x\n", id[3], id
               [4]);

    /* 方式2：使用 rt_spi_transfer_message()发送命令读取ID */
    struct rt_spi_message msg1, msg2;

    msg1.send_buf    = &w25x_read_id;
    msg1.recv_buf    = RT_NULL;
    msg1.length      = 1;
    msg1.cs_take     = 1;
    msg1.cs_release   = 0;
    msg1.next        = &msg2;

    msg2.send_buf    = RT_NULL;
    msg2.recv_buf    = id;
    msg2.length      = 5;
    msg2.cs_take     = 0;
    msg2.cs_release   = 1;
    msg2.next        = RT_NULL;

    rt_spi_transfer_message(spi_dev_w25q, &msg1);
    rt_kprintf("use rt_spi_transfer_message() read w25q ID is:%x%x\n", id[3], id
               [4]);
}

/*
 * 导出到 msh 命令列表中
 */
MSH_CMD_EXPORT(spi_w25q_sample, spi w25q sample);
```

第 20 章

WATCHDOG 设备

20.1 WATCHDOG 简介

硬件看门狗（watchdog timer）是一个定时器，其定时输出连接到电路的复位端。在产品化的嵌入式系统中，为了使系统在异常情况下能自动复位，一般都需要引入看门狗。

当看门狗启动后，计数器开始自动计数，在计数器溢出前如果没有被复位，计数器溢出就会对 CPU 产生一个复位信号使系统重启（俗称“被狗咬”）。系统正常运行时，需要在看门狗允许的时间间隔内对看门狗计数器清零（俗称“喂狗”），不让复位信号产生。如果系统不出问题，程序能够按时“喂狗”。一旦程序跑飞，没有“喂狗”，系统“被咬”复位。

一般情况下可以在 RT-Thread 的 idle 回调函数和关键任务里喂狗。

20.2 访问看门狗设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问看门狗硬件，相关接口如下所示：

函数	描述
rt_device_find()	根据看门狗设备设备名称查找设备获取设备句柄
rt_device_init()	初始化看门狗设备
rt_device_control()	控制看门狗设备
rt_device_close()	关闭看门狗设备

20.2.1 查找看门狗

应用程序根据看门狗设备名称获取设备句柄，进而可以操作看门狗设备，查找设备函数如下所示：

```
rt_device_t rt_device_find(const char* name);
```

参数	描述
name	看门狗设备名称
返回	—
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

使用示例如下所示：

```
#define WDT_DEVICE_NAME      "wdt"      /* 看门狗设备名称 */

static rt_device_t wdg_dev;          /* 看门狗设备句柄 */
/* 根据设备名称查找看门狗设备，获取设备句柄 */
wdg_dev = rt_device_find(WDT_DEVICE_NAME);
```

20.2.2 初始化看门狗

使用看门狗设备前需要先初始化，通过如下函数初始化看门狗设备：

```
rt_err_t rt_device_init(rt_device_t dev);
```

参数	描述
dev	看门狗设备句柄
返回	—
RT_EOK	设备初始化成功
-RT_ENOSYS	初始化失败，看门狗设备驱动初始化函数为空
其他错误码	设备打开失败

使用示例如下所示：

```
#define WDT_DEVICE_NAME      "wdt"      /* 看门狗设备名称 */

static rt_device_t wdg_dev;          /* 看门狗设备句柄 */
/* 根据设备名称查找看门狗设备，获取设备句柄 */
wdg_dev = rt_device_find(WDT_DEVICE_NAME);

/* 初始化设备 */
rt_device_init(wdg_dev);
```

20.2.3 控制看门狗

通过命令控制字，应用程序可以对看门狗设备进行配置，通过如下函数完成：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
```

参数	描述
dev	看门狗设备句柄
cmd	命令控制字
arg	控制的参数
返回	—
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

命令控制字可取如下宏定义值：

#define RT_DEVICE_CTRL_WDT_GET_TIMEOUT	(1) /* 获取溢出时间 */
#define RT_DEVICE_CTRL_WDT_SET_TIMEOUT	(2) /* 设置溢出时间 */
#define RT_DEVICE_CTRL_WDT_GET_TIMELEFT	(3) /* 获取剩余时间 */
#define RT_DEVICE_CTRL_WDT_KEEPALIVE	(4) /* 喂狗 */
#define RT_DEVICE_CTRL_WDT_START	(5) /* 启动看门狗 */
#define RT_DEVICE_CTRL_WDT_STOP	(6) /* 停止看门狗 */

设置看门狗溢出时间使用示例如下所示：

```
#define WDT_DEVICE_NAME      "wdt"      /* 看门狗设备名称 */

rt_uint32_t timeout = 1;           /* 溢出时间，单位：秒*/
static rt_device_t wdg_dev;       /* 看门狗设备句柄 */
/* 根据设备名称查找看门狗设备，获取设备句柄 */
wdg_dev = rt_device_find(WDT_DEVICE_NAME);
/* 初始化设备 */
rt_device_init(wdg_dev);

/* 设置看门狗溢出时间 */
rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_SET_TIMEOUT, (void *)timeout);
/* 设置空闲线程回调函数 */
rt_thread_idle_sethook(idle_hook);
```

在空闲线程钩子函数里喂狗使用示例如下所示：

```
static void idle_hook(void)
{
    /* 在空闲线程的回调函数里喂狗 */
```

```
rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
}
```

20.2.4 关闭看门狗

当应用程序完成看门狗操作后，可以关闭看门狗设备，通过如下函数完成：

```
rt_err_t rt_device_close(rt_device_t dev);
```

参数	描述
dev	看门狗设备句柄
返回	—
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

关闭设备接口和打开设备接口需配对使用，打开一次设备对应要关闭一次设备，这样设备才会被完全关闭，否则设备仍处于未关闭状态。

20.3 看门狗设备使用示例

看门狗设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 根据设备名称“wdt”查找设备获取设备句柄。
2. 初始化设备后设置看门狗溢出时间。
3. 启动看门狗。
4. 喂狗：设置空闲线程回调函数，在空闲线程回调函数中喂狗。

```
/*
 * 程序清单：这是一个独立看门狗设备使用例程
 * 例程导出了 wdt_sample 命令到控制终端
 * 命令调用格式：wdt_sample wdt
 * 命令解释：命令第二个参数是要使用的看门狗设备名称，为空则使用例程默认的看门狗设备。
 * 程序功能：程序通过设备名称查找看门狗设备，然后初始化设备并设置看门狗设备溢出时间。
 *           然后设置空闲线程回调函数，在回调函数里会喂狗。
*/
#include <rtthread.h>
#include <rtdevice.h>
```

```
#define WDT_DEVICE_NAME      "wdt"      /* 看门狗设备名称 */

static rt_device_t wdg_dev;           /* 看门狗设备句柄 */

static void idle_hook(void)
{
    /* 在空闲线程的回调函数里喂狗 */
    rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
    rt_kprintf("feed the dog!\n");
}

static int wdt_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    rt_uint32_t timeout = 1;          /* 溢出时间，单位：秒 */
    char device_name[RT_NAME_MAX];

    /* 判断命令行参数是否给定了设备名称 */
    if (argc == 2)
    {
        rt_strncpy(device_name, argv[1], RT_NAME_MAX);
    }
    else
    {
        rt_strncpy(device_name, WDT_DEVICE_NAME, RT_NAME_MAX);
    }

    /* 根据设备名称查找看门狗设备，获取设备句柄 */
    wdg_dev = rt_device_find(device_name);
    if (!wdg_dev)
    {
        rt_kprintf("find %s failed!\n", device_name);
        return RT_ERROR;
    }

    /* 设置看门狗溢出时间 */
    ret = rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_SET_TIMEOUT, &timeout);
    if (ret != RT_EOK)
    {
        rt_kprintf("set %s timeout failed!\n", device_name);
        return RT_ERROR;
    }

    /* 启动看门狗 */
    ret = rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_START, RT_NULL);
    if (ret != RT_EOK)
    {
        rt_kprintf("start %s failed!\n", device_name);
        return -RT_ERROR;
    }

    /* 设置空闲线程回调函数 */
}
```

```
rt_thread_idle_sethook(idle_hook);

return ret;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(wdt_sample, wdt_sample);
```

第 21 章

WLAN 设备

随着物联网快速发展，越来越多的嵌入式设备上搭载了 WIFI 无线网络设备。为了能够管理 WIFI 网络设备，RT-Thread 引入了 WLAN 设备管理框架。这套框架具备控制和管理 WIFI 的众多功能，为开发者使用 WIFI 设备提供许多便利。

21.1 WLAN 框架简介

WLAN 框架是 RT-Thread 开发的一套用于管理 WIFI 的中间件。对下连接具体的 WIFI 驱动，控制 WIFI 的连接断开，扫描等操作。对上承载不同的应用，为应用提供 WIFI 控制，事件，数据导流等操作，为上层应用提供统一的 WIFI 控制接口。WLAN 框架主要由三个部分组成。DEV 驱动接口层，为 WLAN 框架提供统一的调用接口。Manage 管理层为用户提供 WIFI 扫描，连接，断线重连等具体功能。Protocol 协议负责处理 WIFI 上产生的数据流，可根据不同的使用场景挂载不同通讯协议，如 LWIP 等。具有使用简单，功能齐全，对接方便，兼容性强等特点。

下图是 WIFI 框架层次图：

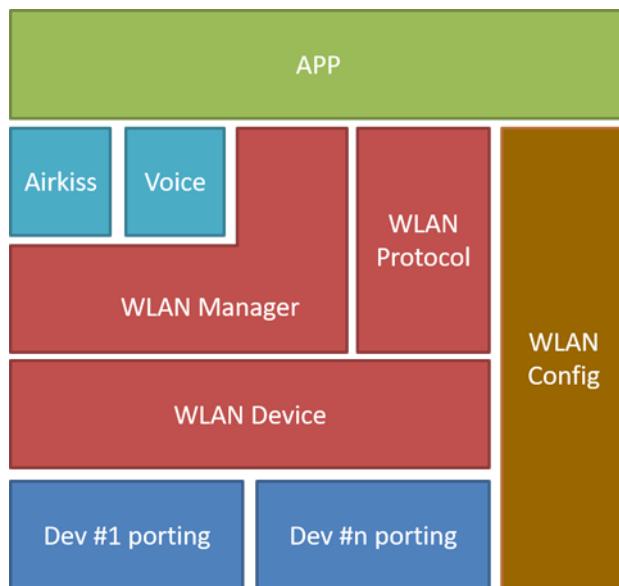


图 21.1: WIFI 框架

第一部分 app 为应用层。是基于 WLAN 框架的具体应用，如 WiFi 相关的 Shell 命令。

第二部分 airkiss、voice 为配网层。提供无线配网和声波配网等功能。

第三部分 WLAN manager 为 WLAN 管理层。能够对 WLAN 设备进行控制和管理。具备设置模式、连接热点、断开热点、启动热点、扫描热点等 WLAN 控制相关的功能。还提供断线重连，自动切换热点等管理功能。

第四部分 WLAN protocol 为协议层。将数据流递交给具体协议进行解析，用户可以指定使用不同的协议进行通信。

第五部分 WLAN config 为参数管理层。管理连接成功的热点信息及密码，并写入非易失的存储介质中。

第六部分 WLAN dev 为驱动接口层。对接具体 WLAN 硬件，为管理层提供统一的调用接口。

21.1.1 功能简介

- 自动连接：打开自动连接功能后，只要 WIFI 处在断线状态，就会自动读取之前连接成功的热点信息，连接热点。如果一个热点连接失败，则切换下一个热点信息进行连接，直到连接成功为止。自动连接使用的热点信息，按连接成功的时间顺序，依次尝试，优先使用最近连接成功的热点信息。连接成功后，将热点信息缓存在最前面，下次断线优先使用。
- 参数存储：存储连接成功的 WIFI 参数，WIFI 参数会在内存中缓存一份，如果配置外部非易失存储接口，则会在外部存储介质中存储一份。用户可根据自己的实际情况，实现 `struct rt_wlan_cfg_ops` 这个结构体，将参数保存任何地方。缓存的参数主要给自动连接提供热点信息，wifi 处在未连接状态时，会读取缓存的参数，尝试连接。
- WIFI 控制：提供完备的 WIFI 控制接口，扫描，连接，热点等。提供 WIFI 相关状态回调事件，断开，连接，连接失败等。为用户提供简单易用的 WIFI 管理接口。
- Shell 命令：可在 Msh 中输入命令控制 WIFI 执行扫描，连接，断开等动作。打印 WIFI 状态等调试信息。

21.1.2 配置选项

在 ENV 工具中使用 `menuconfig` 命令按照以下菜单进入 WLAN 配置界面：

```
RT-Thread Components -> Device Drivers -> Using WiFi ->
```

各个配置选项详细描述如下：

<code>(wlan0) The device name for station</code>	<code>/* Station 设备默认名字 */</code>
<code>名字 */</code>	
<code>(wlan1) The device name for ap</code>	<code>/* AP 设备默认名字 */</code>
<code>*/</code>	
<code>(32) SSID maximum length</code>	<code>/* SSID 最大长度 */</code>
<code>(64) Password maximum length</code>	<code>/* 密码最大长度 */</code>
<code>(2) Driver events maxcount</code>	<code>/* 事件最大注册数 */</code>
<code>[*] Connection management Enable</code>	<code>/* 连接管理功能使能 */</code>
<code>*/</code>	

```

(10000) Set scan timeout time(ms)          /* 扫描超时时间 */
(10000) Set connect timeout time(ms)       /* 连接超时时间 */
[*]      Automatic sorting of scan results /* 扫描结果自动排序 */
*/
[*]      MSH command Enable                /* MSH 命令功能使能 */
*/
[*]      Auto connect Enable               /* 自动连接功能使能 */
*/
(2000)   Auto connect period(ms)          /* 断线检查周期 */
-* WiFi information automatically saved Enable
使能 */
(3)     Maximum number of WiFi information automatically saved /* 连接参数最大存储
数量 */
[*]     Transport protocol manage Enable  /* 传输协议管理功能
使能 */
(8)     Transport protocol name length   /* 传输协议名字最大
长度 */
(2)     Transport protocol maxcount      /* 传输协议类型数量 */
*/
(lwip)  Default transport protocol        /* 默认传输协议名字 */
*/
[*]     LWIP transport protocol Enable    /* LWIP 传输协议使能 */
*/
(lwip)  LWIP transport protocol name     /* LWIP 传输协议名字 */
*/
[ ]     Forced use of PBUF transmission  /* 强制使用 PBUF 交
换数据 */
-* WLAN work queue thread Enable         /* WLAN 线程使能 */
(wlan)  WLAN work queue thread name      /* WLAN 线程名字 */
(2048)  WLAN work queue thread size     /* WLAN 线程栈大小 */
*/
(15)    WLAN work queue thread priority  /* WLAN 线程优先级 */
*/
[ ]     Enable WLAN Debugging Options   ---> /* 打开调试日志 */

```

应用程序通过 WLAN 连接管理层相关 API 来访问硬件设备，相关接口如下所示：

21.2 WLAN 初始化

函数	描述
rt_wlan_init()	初始化连接管理器
rt_wlan_set_mode()	设置工作模式
rt_wlan_get_mode()	获取设备工作模式

21.2.1 连接管理初始化

```
int rt_wlan_init(void)
```

初始化连接管理器需要的静态资源，如全局变量，线程，互斥锁等。支持自动初始化，无需用户调用。如果没用使能自动初始化，在使用 WLAN 相关 API 之前，需要手动调用进行初始化。

参数	描述
无	
返回值	描述
0	执行成功

21.2.2 设置设备模式

```
rt_err_t rt_wlan_set_mode(const char *dev_name, rt_wlan_mode_t mode)
```

设置 WLAN 设备的工作模。同一个设备，切换相同的模式无效，一种模式，只能存在一个设备，不能两个设备设置同一个模式。一般的，一个设备只支持一种模式。

参数	描述
dev_name	设备名字
mode	工作模式
返回值	描述
RT_EOK	设置成功
-RT_ERROR	设置失败

WLAN 设备工作模式如下：

```
typedef enum
{
    RT_WLAN_NONE,                      /* 停止工作模式 */
    RT_WLAN_STATION,                   /* 无线终端模式 */
    RT_WLAN_AP,                        /* 无线接入服务模式 */
    RT_WLAN_MODE_MAX                  /* 无效 */
} rt_wlan_mode_t;
```

系统默认会提供一个默认的 STA 设备名和 AP 设备名，下面示例将展示默认 STA 设备工作在无线终端模式：

```
rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
```

21.2.3 获取设备模式

```
rt_wlan_mode_t rt_wlan_get_mode(const char *dev_name)
```

获得设备的工作模式。

参数	描述
dev_name	设备名字
返回值	描述
RT_WLAN_NONE	设备停止工作
RT_WLAN_STATION	无线终端模式
RT_WLAN_AP	无线接入服务模式

21.3 WLAN 连接

函数	描述
rt_wlan_connect()	连接热点
rt_wlan_connect_adv()	无阻塞连接热点
rt_wlan_disconnect()	断开热点
rt_wlan_is_connected()	获取连接标志
rt_wlan_is_ready()	获取就绪标志
rt_wlan_get_info()	获取连接信息
rt_wlan_get_rssi()	获取信号强度

21.3.1 连接热点

```
rt_err_t rt_wlan_connect(const char *ssid, const char *password)
```

阻塞式连接热点。此 API 调用的时间会比较长，连接成功或失败后才会返回。

参数	描述
ssid	热点的名字
password	热点密码，无密码传空
返回	描述
RT_EOK	连接成功
-RT_ERROR	连接失败

WLAN 连接成功，还不能进行数据通讯，需要等待连接就绪才能通讯。

21.3.2 无阻塞连接

```
rt_err_t rt_wlan_connect_adv(struct rt_wlan_info *info, const char *password)
```

非阻塞连接热点，连接参数可通过扫描获得或手动指定。一般用于连接特定热点或隐藏热点，返回值仅表示连接动作是否开始执行，是否连接成功需要主动查询或设置回调通知。

参数	描述
info	连接信息
password	热点密码，无密码传空
返回	描述
RT_EOK	执行成功
-RT_ERROR	执行失败

连接信息必须配置的项有`security`、`ssid`。完整配置如下：

```
struct rt_wlan_info
{
    rt_wlan_security_t security;           /* 安全类型 */
    rt_802_11_band_t band;                /* 2.4G / 5G */
    rt_uint32_t datarate;                 /* 连接速率 */
    rt_int16_t channel;                  /* 通道 */
    rt_int16_t rssi;                     /* 信号强度 */
    rt_wlan_ssid_t ssid;                 /* 热点名称 */
    rt_uint8_t bssid[RT_WLAN_BSSID_MAX_LENGTH]; /* 热点物理地址 */
    rt_uint8_t hidden;                   /* 热点隐藏标志 */
};
```

安全模式如下所示：

```
typedef enum
{
    SECURITY_OPEN          = 0,                      /* Open
    security               */
    SECURITY_WEP_PSK        = WEP_ENABLED,           /* WEP
    Security with open authentication */
    SECURITY_WEP_SHARED     = (WEP_ENABLED | SHARED_ENABLED), /* WEP
    Security with shared authentication */
    SECURITY_WPA_TKIP_PSK    = (WPA_SECURITY | TKIP_ENABLED), /* WPA
    Security with TKIP */
    SECURITY_WPA_AES_PSK    = (WPA_SECURITY | AES_ENABLED), /* WPA
    Security with AES */
    SECURITY_WPA2_AES_PSK   = (WPA2_SECURITY | AES_ENABLED), /* WPA2
    Security with AES */
};
```

```

SECURITY_WPA2_TKIP_PSK = (WPA2_SECURITY | TKIP_ENABLED),           /* WPA2
    Security with TKIP
SECURITY_WPA2_MIXED_PSK = (WPA2_SECURITY | AES_ENABLED | TKIP_ENABLED), /* WPA2
    Security with AES & TKIP
SECURITY_WPS_OPEN        = WPS_ENABLED,                                /* WPS
    with open security
SECURITY_WPS_SECURE       = (WPS_ENABLED | AES_ENABLED),           /* WPS
    with AES security
SECURITY_UNKNOWN          = -1,                                       /*
    security is unknown.
} rt_wlan_security_t;

```

下面将展示使用指定连接信息执行连接。

```

struct rt_wlan_info info;

INVALID_INFO(&info);                                     /* 初始化 info */
SSID_SET(&info, "test_ap");                            /* 设置热点名字 */
info.security = SECURITY_WPA2_AES_PSK;                 /* 指定安全类型 */
rt_wlan_connect_adv(&info, "12345678");              /* 执行连接动作 */
while (rt_wlan_is_connected() == RT_FALSE);             /* 等待连接成功 */

```

21.3.3 断开热点

`rt_err_t rt_wlan_disconnect(void)`

阻塞式断开连接，返回值表示是否成功断开。

参数	描述
无	
返回	描述
RT_EOK	断开成功
-RT_ERROR	断开失败

执行断开之前，建议先查询是否已经连接，如果已经连接，再执行断开。示例代码如下：

```

if (rt_wlan_is_connected())                               /* 判断是否已经连接 */
{
    rt_wlan_disconnect();                             /* 断开连接 */
}

```

21.3.4 获取连接标志

`rt_bool_t rt_wlan_is_connected(void)`

查询是否连接到热点。

参数	描述
无	
返回	描述
RT_TRUE	已经连接
RT_FALSE	没有连接

21.3.5 获取就绪标志

```
rt_bool_t rt_wlan_is_ready(void)
```

查询连接是否就绪。一般的，获取到 IP 表示已经准备就绪，可以传输数据。

参数	描述
无	
返回	描述
RT_TRUE	已经就绪
RT_FALSE	没有就绪

21.3.6 获取连接信息

```
rt_err_t rt_wlan_get_info(struct rt_wlan_info *info)
```

获取详细的连接信息，可获取热点名字、通道、信号强度、安全类型等。

参数	描述
info	info 对象
返回	描述
RT_EOK	获取成功
-RT_ERROR	获取失败

21.3.7 获取信号强度

```
int rt_wlan_get_rssi(void);
```

获得信号强度。信号强度为负值，值越大信号越强。例如信号强度 -25 比信号强度 -55 要好。

参数	描述
无	
返回	描述
负数	信号强度
0	未连接

21.4 WLAN 扫描

函数	描述
rt_wlan_scan()	异步扫描
rt_wlan_scan_sync()	同步扫描
rt_wlan_scan_with_info()	条件扫描
rt_wlan_scan_get_info_num()	获取热点个数
rt_wlan_scan_get_info()	拷贝热点信息
rt_wlan_scan_get_result()	获取扫描缓存
rt_wlan_scan_result_clean()	清理扫描缓存
rt_wlan_find_best_by_cache()	查找最佳热点

21.4.1 异步扫描

```
rt_err_t rt_wlan_scan(void)
```

异步扫描函数，扫描完成需要通过回调进行通知。

参数	描述
无	
返回	描述
RT_EOK	启动扫描成功
-RT_ERROR	启动扫描失败

21.4.2 同步扫描

```
struct rt_wlan_scan_result *rt_wlan_scan_sync(void)
```

同步扫描函数，扫描全部热点信息，完成过直接返回扫描结果。

参数	描述
无	
返回	描述
扫描结果	热点信息和数量
RT_NULL	扫描失败

该接口执行成功，会返回 `struct rt_wlan_scan_result` 类型的指针，包含了热点的详细信息和数量。结构体定义如下：

```
struct rt_wlan_scan_result
{
    rt_int32_t num;                                /* 热点个数 */
    struct rt_wlan_info *info;                      /* 热点信息 */
};
```

结构体中的 `info` 是连续的内存块，可通过类似数组的形式进行访问。示例如下：

```
result = rt_wlan_scan_sync(void);                  /* 获取扫描结果 */
for (i = 0; i < result->num; i++)                /* 根据扫描到的结果，进行遍历 */
{
    printf("SSID:%s\n", result->info[i].ssid.val); /* 使用数组的形式进行访问，打印
                                                扫描到的 SSID 信息 */
}
```

21.4.3 条件扫描

```
struct rt_wlan_scan_result *rt_wlan_scan_with_info(struct rt_wlan_info *info)
```

同步条件扫描。根据参入的条件进行过滤，可用于扫描指定 SSID。

参数	描述
info	通过 <code>info</code> 指定限定条件
返回	描述
扫描结果	热点信息和数量
RT_NULL	扫描失败

下面示例将展示扫描指定 SSID 的热点信息。

```
struct rt_wlan_info info;
INVALID_INFO(&info);                                /* 初始化 info */
SSID_SET(&info, "test_ap");                         /* 指定 SSID */
```

```
result = rt_wlan_scan_with_info(&info);           /* 开始同步扫描 */
```

21.4.4 获取热点个数

```
int rt_wlan_scan_get_result_num(void)
```

返回扫描到的热点数量。

参数	描述
无	
返回	描述
数量	热点数量

21.4.5 拷贝热点信息

```
int rt_wlan_scan_get_info(struct rt_wlan_info *info, int num)
```

拷贝热点信息。

参数	描述
info	info 缓存，用于保存拷贝结果
num	info 个数
返回	描述
数量	实际拷贝的个数

下面代码片段将展示如何拷贝热点信息

```
num = rt_wlan_scan_get_result_num();           /* 查询热点数量 */
info = rt_malloc(sizeof(struct rt_wlan_info) * num); /* 分配内存 */
rt_wlan_scan_get_info(info, num);               /* 拷贝 */
```

21.4.6 获取扫描缓存

```
struct rt_wlan_scan_result *rt_wlan_scan_get_result(void)
```

返回扫描缓存。

参数	描述
无	
返回	描述
扫描缓存指针	该指针不安全，仅作临时访问

参数	描述
----	----

21.4.7 清理扫描缓存

```
void rt_wlan_scan_result_clean(void)
```

清理扫描缓存。

参数	描述
无	
返回	描述
无	

21.4.8 查找最佳热点

```
rt_bool_t rt_wlan_find_best_by_cache(const char *ssid, struct rt_wlan_info *info)
```

指定 SSID，在扫描缓存中查找信号最好的热点信息。

参数	描述
ssid	指定需要查询的 ssid
info	存查询到的热点信息
返回	描述
RT_FALSE	没有查到
RT_TRUE	查到

21.5 WLAN 热点

函数	描述
rt_wlan_start_ap()	启动热点
rt_wlan_start_ap_adv()	无阻塞启动热点
rt_wlan_ap_is_active()	获取启动标志
rt_wlan_ap_stop()	停止热点
rt_wlan_ap_get_info()	获取热点信息

21.5.1 启动热点

```
rt_err_t rt_wlan_start_ap(const char *ssid, const char *password)
```

阻塞式启动热点，返回值表示是否启动成功。

参数	描述
ssid	热点名字
password	热点密码。
返回	描述
RT_EOK	启动成功
-RT_ERROR	启动失败

21.5.2 非阻塞启动热点

```
rt_err_t rt_wlan_start_ap_adv(struct rt_wlan_info *info, const char *password)
```

非阻塞启动热点，可以指加密类型，通道等。热点是否启动需要手动查询或回调通知。

参数	描述
info	热点信息
password	热点密码，开放热点传空
返回	描述
RT_EOK	执行成功
-RT_ERROR	执行失败

21.5.3 获取启动标志

```
rt_bool_t rt_wlan_ap_is_active(void)
```

查询热点是否处于活动状态。

参数	描述
无	
返回	描述
RT_TRUE	热点启动
-RT_FALSE	热点未启动

21.5.4 停止热点

```
rt_err_t rt_wlan_ap_stop(void)
```

阻塞式停止热点。停止之前先查询是否已经启动，已经启动后在停止。

参数	描述
无	
返回	描述
RT_EOK	停止成功
-RT_ERROR	停止失败

21.5.5 获取热点信息

```
rt_err_t rt_wlan_ap_get_info(struct rt_wlan_info *info)
```

获取热点相关信息，如热点名字，通道等。

参数	描述
info	热点信息
返回	描述
RT_EOK	获取成功
-RT_ERROR	获取失败

21.6 WLAN 自动重连

函数	描述
rt_wlan_config_autoreconnect()	启动/停止自动重连
rt_wlan_get_autoreconnect_mode()	获取自动重连模式

21.6.1 启动/停止自动重连

```
void rt_wlan_config_autoreconnect(rt_bool_t enable)
```

开启或关闭自动重连模式，当没有网络时，会自动进行重连。

参数	描述
<code>enable</code>	开启或关闭
返回	描述
无	执行成功

21.6.2 获取自动重连模式

```
rt_bool_t rt_wlan_get_autoreconnect_mode(void)
```

查询自动重连是否启动。

参数	描述
无	
返回	描述
<code>RT_TRUE</code>	启动
<code>RT_FALSE</code>	未启动

21.7 WLAN 事件回调

函数	描述
<code>rt_wlan_register_event_handler()</code>	事件注册
<code>rt_wlan_unregister_event_handler()</code>	解除注册

21.7.1 事件注册

```
rt_err_t rt_wlan_register_event_handler(rt_wlan_event_t event, rt_wlan_event_handler
handler, void *parameter)
```

注册事件回调函数。

参数	描述
<code>event</code>	事件类型
<code>handler</code>	事件处理函数
<code>parameter</code>	用户参数
返回	描述

参数	描述
RT_EOK	注册成功
-RT_ERROR	注册失败

WLAN 产生如下事件时，触发回调：

```
typedef enum
{
    RT_WLAN_EVT_READY = 0,           /* 网络就绪 */
    RT_WLAN_EVT_SCAN_DONE,          /* 扫描完成 */
    RT_WLAN_EVT_SCAN_REPORT,        /* 扫描到一个热点 */
    RT_WLAN_EVT_STA_CONNECTED,      /* 连接成功 */
    RT_WLAN_EVT_STA_CONNECTED_FAIL, /* 连接失败 */
    RT_WLAN_EVT_STA_DISCONNECTED,   /* 断开连接 */
    RT_WLAN_EVT_AP_START,          /* 热点启动 */
    RT_WLAN_EVT_AP_STOP,           /* 热点停止 */
    RT_WLAN_EVT_AP_ASSOCIATED,      /* STA 接入 */
    RT_WLAN_EVT_AP_DISASSOCIATED,   /* STA 断开 */
} rt_wlan_event_t;
```

WLAN 回调函数定义为：`void (*rt_wlan_event_handler)(int event, struct rt_wlan_buff *buff, void *parameter);`，其中 `buff` 根据事件的不同有不同的涵义。详细信息参考下表。

事件	类型	描述
RT_WLAN_EVT_READY	ip_addr_t *	IP 地址
RT_WLAN_EVT_SCAN_DONE	struct rt_wlan_scan_result *	扫描的结果
RT_WLAN_EVT_SCAN_REPORT	struct rt_wlan_info *	扫描到的热点信息
RT_WLAN_EVT_STA_CONNECTED	struct rt_wlan_info *	连接成功的 Station 信息
RT_WLAN_EVT_STA_CONNECTED_	struct rt_wlan_info *	连接失败的 Station 信息
RT_WLAN_EVT_STA_DISCONNECTED	struct rt_wlan_info *	断开连接的 Station 信息
RT_WLAN_EVT_AP_START	struct rt_wlan_info *	启动成功的 AP 信息
RT_WLAN_EVT_AP_STOP	struct rt_wlan_info *	启动失败的 AP 信息
RT_WLAN_EVT_AP_ASSOCIATED	struct rt_wlan_info *	连入的 Station 信息
RT_WLAN_EVT_AP_DISASSOCIATED	struct rt_wlan_info *	断开的 Station 信息

21.7.2 解除注册

```
rt_err_t rt_wlan_unregister_event_handler(rt_wlan_event_t event)
```

事件解除注册。

参数	描述
event	事件类型
返回	描述
RT_EOK	解除成功

21.8 WLAN 功耗管理

函数	描述
rt_wlan_set_powersave()	设置功耗等级
rt_wlan_get_powersave()	获取功耗等级

21.8.1 设置功耗等级

设置功耗等级，用于 station 模式。

```
rt_err_t rt_wlan_set_powersave(int level)
```

参数	描述
level	功耗等级
返回	描述
RT_EOK	设置成功
-RT_ERROR	设置失败

21.8.2 获取功耗等级

```
int rt_wlan_get_powersave(void)
```

获取当前工作功耗等级。

参数	描述
无	
返回	描述
功耗级别	

21.9 FinSH 命令

使用 shell 命令，可以帮助我们快速调试 WiFi 相关功能。wifi 相关的 shell 命令如下：

wifi	/* 打印帮助 */
wifi help	/* 查看帮助 */
wifi join SSID [PASSWORD]	/* 连接 wifi, SSID 为空, 使用配置自动连接 */
wifi ap SSID [PASSWORD]	/* 建立热点 */
wifi scan	/* 扫描全部热点 */
wifi disc	/* 断开连接 */
wifi ap_stop	/* 停止热点 */
wifi status	/* 打印 wifi 状态 sta + ap */
wifi smartconfig	/* 启动配网功能 */

21.9.1 WiFi 扫描

wifi 扫描命令为 `wifi scan`，执行 `wifi scan` 命令后，会将周围的热点信息打印在终端上。通过打印的热点信息，可以看到 SSID，MAC 地址等多项属性。

在 msh 中输入该命令，扫描结果如下所示：

wifi scan	SSID	MAC	security	rssi	chn	Mbps
<hr/>						
rtt_test_ssid_1	c0:3d:46:00:3e:aa	OPEN	-14	8	300	
test_ssid	3c:f5:91:8e:4c:79	WPA2_AES_PSK	-18	6	72	
rtt_test_ssid_2	ec:88:8f:88:aa:9a	WPA2_MIXED_PSK	-47	6	144	
rtt_test_ssid_3	c0:3d:46:00:41:ca	WPA2_MIXED_PSK	-48	3	300	

21.9.2 WiFi 连接

wifi 扫描命令为 `wifi join`，命令后面需要跟热点名称和热点密码，没有密码可不输入这一项。执行 WiFi 连接命令后，如果热点存在，且密码正确，开发板会连接上热点，并获得 IP 地址。网络连接成功后，可使用 `socket` 套接字进行网络通讯。

wifi 连接命令使用示例如下所示，连接成功后，将在终端上打印获得的 IP 地址，如下所示：

```
wifi join ssid_test 12345678
[I/WLAN.mgnt] wifi connect success ssid:ssid_test
[I/WLAN.lwip] Got IP address : 192.168.1.110
```

21.9.3 WiFi 断开

wifi 断开的命令为 `wifi disc`，执行 WiFi 断开命令后，开发板将断开与热点的连接。

WiFi 断开命令使用示例如下所示，断开成功后，将在终端上打印如下信息，如下所示

```
wifi disc  
[I/WLAN.mgnt] disconnect success!
```

21.10 WLAN 设备使用示例

21.10.1 扫描示例

下面这段代码将展示 WiFi 同步扫描，然后我们将结果打印在终端上。先需要执行 WIFI 初始化，然后执行 WIFI 扫描函数 `rt_wlan_scan_sync`，这个函数是同步的，函数返回的扫描的数量和结果。在这个示例中，会将扫描的热点名字打印出来。

```
#include <rthw.h>  
#include <rtthread.h>  
  
#include <wlan_mgnt.h>  
#include <wlan_prot.h>  
#include <wlan_cfg.h>  
  
void wifi_scan(void)  
{  
    struct rt_wlan_scan_result *result;  
    int i = 0;  
  
    /* Configuring WLAN device working mode */  
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);  
    /* WiFi scan */  
    result = rt_wlan_scan_sync();  
    /* Print scan results */  
    rt_kprintf("scan num:%d\n", result->num);  
    for (i = 0; i < result->num; i++)  
    {  
        rt_kprintf("ssid:%s\n", result->info[i].ssid.val);  
    }  
}  
  
int scan(int argc, char *argv[]){  
    wifi_scan();  
    return 0;  
}  
MSH_CMD_EXPORT(scan, scan test.);
```

运行结果如下：

```

\ | /
- RT - Thread Operating System
/ | \ 3.1.0 build Sep 11 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[SFUD] w25ql28 flash device is initialize success.
msh />[I/FAL] RT-Thread Flash Abstraction Layer (V0.2.0) initialize success.
[I/OTA] RT-Thread OTA package(V0.1.3) initialize success.
[I/OTA] Verify 'wifi_image' partition(fw ver: 1.0, timestamp: 1529386280) success.
[I/WICED] wifi initialize done!
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />scan ← 扫描测试命令
scan num:3 ← 扫描到的总数量
ssid:SSID-A ← 扫描到的热点名字
ssid:SSID-B
ssid:YST2016
msh />

```

图 21.2: 扫描

21.10.2 连接与断开示例

下面这段代码将展示 WiFi 同步连接。需要先执行 WIFI 初始化，然后创建一个用于等待 `RT_WLAN_EVT_READY` 事件的信号量。注册需要关注的事件的回调函数，执行 `rt_wlan_connect` wifi 连接函数，函数返回表示是否已经连接成功。但是连接成功还不能进行通信，还需要等待网络获取 IP。使用事先创建的信号量等待网络准备好，网络准备好后，就能正常通信了。

连接上 WiFi 后，等待一段时间后，执行 `rt_wlan_disconnect` 函数断开连接。断开操作是阻塞的，返回值表示是否断开成功。

```

#include <rthw.h>
#include <rtthread.h>

#include <wlan_mgnt.h>
#include <wlan_prot.h>
#include <wlan_cfg.h>

#define WLAN_SSID          "SSID-A"
#define WLAN_PASSWORD       "12345678"
#define NET_READY_TIME_OUT (rt_tick_from_millisecond(15 * 1000))

static rt_sem_t net_ready = RT_NULL;

static void
wifi_ready_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    rt_sem_release(net_ready);
}

static void

```

```
wifi_connect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_disconnect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_connect_fail_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

rt_err_t wifi_connect(void)
{
    rt_err_t result = RT_EOK;

    /* Configuring WLAN device working mode */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* station connect */
    rt_kprintf("start to connect ap ... \n");
    net_ready = rt_sem_create("net_ready", 0, RT_IPC_FLAG_FIFO);
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY,
        wifi_ready_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED,
        wifi_connect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
        wifi_disconnect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED_FAIL,
        wifi_connect_fail_callback, RT_NULL);

    /* connect wifi */
    result = rt_wlan_connect(WLAN_SSID, WLAN_PASSWORD);
```

```
if (result == RT_EOK)
{
    /* waiting for IP to be got successfully */
    result = rt_sem_take(net_ready, NET_READY_TIME_OUT);
    if (result == RT_EOK)
    {
        rt_kprintf("networking ready!\n");
    }
    else
    {
        rt_kprintf("wait ip got timeout!\n");
    }
    rt_wlan_unregister_event_handler(RT_WLAN_EVT_READY);
    rt_sem_delete(net_ready);

    rt_thread_delay(rt_tick_from_millisecond(5 * 1000));
    rt_kprintf("wifi disconnect test!\n");
    /* disconnect */
    result = rt_wlan_disconnect();
    if (result != RT_EOK)
    {
        rt_kprintf("disconnect failed\n");
        return result;
    }
    rt_kprintf("disconnect success\n");
}
else
{
    rt_kprintf("connect failed!\n");
}
return result;
}

int connect(int argc, char *argv[])
{
    wifi_connect();
    return 0;
}
MSH_CMD_EXPORT(connect, connect test.);
```

运行结果如下

```

\ | /
- RT -      Thread Operating System
/ | \  3.1.0 build Sep 11 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[SFUD] w25ql28 flash device is initialize success.
msh />[I/FAL] RT-Thread Flash Abstraction Layer (V0.2.0) initialize success.
[I/OTA] RT-Thread OTA package(V0.1.3) initialize success.
[I/OTA] Verify 'wifi_image' partition(fw ver: 1.0, timestamp: 1529386280) success.
[I/WICED] wifi initialize done!
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />connect ← 连接测试命令
start to connect ap ...
join ssid:SSID-A
[I/WLAN.mgnt] wifi connect success ssid:SSID-A ← 连接连接
wifi_connect_callback
wifi_ready_callback
networking ready!
[I/WLAN.lwip] Got IP address : 192.168.43.10 ← 获得IP地址
wifi disconnect test! ← 断开连接
wifi_disconnect_callback
disconnect success ← 断开成功
msh />

```

图 21.3: 连接断开

21.10.3 自动连接示例

先开启自动重连功能，使用命令行连接上一个热点 A 后，在连接上另一个热点 B。等待几秒后，将热点 B 断电，系统会自动重试连接 B 热点，此时 B 热点连接不上，系统自动切换热点 A 进行连接。连接成功 A 后，系统停止连接。

```

#include <rthw.h>
#include <rtthread.h>

#include <wlan_mgnt.h>
#include <wlan_prot.h>
#include <wlan_cfg.h>

static void
wifi_ready_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
}

static void
wifi_connect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

```

```
    }

}

static void
wifi_disconnect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_connect_fail_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

int wifi_autoconnect(void)
{
    /* Configuring WLAN device working mode */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* Start automatic connection */
    rt_wlan_config_autoreconnect(RT_TRUE);
    /* register event */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY,
        wifi_ready_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED,
        wifi_connect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
        wifi_disconnect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED_FAIL,
        wifi_connect_fail_callback, RT_NULL);
    return 0;
}

int auto_connect(int argc, char *argv[])
{
    wifi_autoconnect();
    return 0;
}
MSH_CMD_EXPORT(auto_connect, auto connect test.);
```

运行结果如下:

```
\ | /  
- RT - Thread Operating System  
/ | \ 3.1.0 build Sep 11 2018  
2006 - 2018 Copyright by rt-thread team  
lwIP-2.0.2 initialized!  
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.  
[SFUD] w25ql28 flash device is initialize success.  
msh />[I/FAL] RT-Thread Flash Abstraction Layer (V0.2.0) initialize success.  
[I/OTA] RT-Thread OTA package(V0.1.3) initialize success.  
[I/OTA] Verify 'wifi_image' partition(fw ver: 1.0, timestamp: 1529386280) success.  
[I/WICED] wifi initialize done!  
[I/WLAN.dev] wlan init success  
[I/WLAN.lwip] eth device init ok name:w0  
msh />auto_connect ← 自动连接测试  
msh />wifi join SSID-A 12345678 ← 连接测试热点A  
join ssid:SSID-A  
[I/WLAN.mgnt] wifi connect success ssid:SSID-A  
wifi_connect_callback  
msh />wifi_ready_callback  
[I/WLAN.lwip] Got IP address : 192.168.43.10 ← 连接成功  
msh />wifi join SSID-B 12345678oo ← 连接测试热点B  
join ssid:SSID-B  
[I/WLAN.mgnt] wifi connect success ssid:SSID-B  
wifi_connect_callback  
msh />wifi_ready_callback  
[I/WLAN.lwip] Got IP address : 172.20.10.2  
msh />wifi_disconnect_callback ← 测试热点B断线  
join ssid:SSID-B  
wifi_connect_fail_callback  
join ssid:SSID-A  
[I/WLAN.mgnt] wifi connect success ssid:SSID-A  
wifi_connect_callback  
wifi_ready_callback  
[I/WLAN.lwip] Got IP address : 192.168.43.10 ← 重连热点A  
msh />
```

图 21.4: 自动连接

第 22 章

FinSH 控制台

在计算机发展的早期，图形系统出现之前，没有鼠标，甚至没有键盘。那时候人们如何与计算机交互呢？最早期的计算机使用打孔的纸条向计算机输入命令，编写程序。后来随着计算机的不断发展，显示器、键盘成为计算机的标准配置，但此时的操作系统还不支持图形界面，计算机先驱们开发了一种软件，它接受用户输入的命令，解释之后，传递给操作系统，并将操作系统执行的结果返回给用户。这个程序像一层外壳包裹在操作系统的外面，所以它被称为 shell。

嵌入式设备通常需要将开发板与 PC 机连接起来通讯，常见连接方式包括：串口、USB、以太网、Wi-Fi 等。一个灵活的 shell 也应该支持在多种连接方式上工作。有了 shell，就像在开发者和计算机之间架起了一座沟通的桥梁，开发者能很方便的获取系统的运行情况，并通过命令控制系统的运行。特别是在调试阶段，有了 shell，开发者除了能更快的定位到问题之外，也能利用 shell 调用测试函数，改变测试函数的参数，减少代码的烧录次数，缩短项目的开发时间。

FinSH 是 RT-Thread 的命令行组件（shell），正是基于上面这些考虑而诞生的，FinSH 的发音为 [fmʃ]。读完本章，我们会对 FinSH 的工作方式以及如何导出自己的命令到 FinSH 有更加深入的了解。

22.1 FinSH 简介

FinSH 是 RT-Thread 的命令行组件，提供一套供用户在命令行调用的操作接口，主要用于调试或查看系统信息。它可以使用串口 / 以太网 / USB 等与 PC 机进行通信，硬件拓扑结构如下图所示：

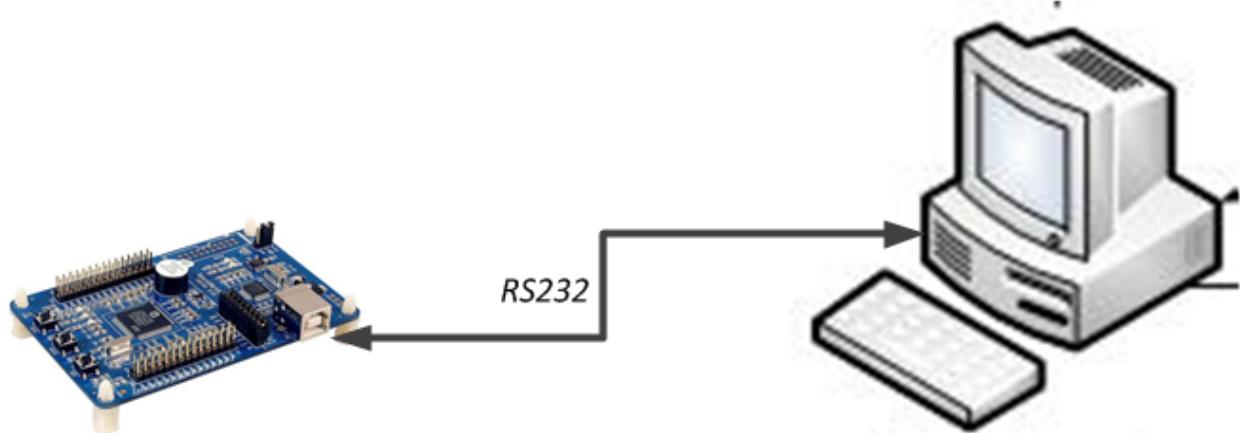


图 22.1: FinSH 硬件连接图

用户在控制终端输入命令，控制终端通过串口、USB、网络等方式将命令传给设备里的 FinSH，FinSH 会读取设备输入命令，解析并自动扫描内部函数表，寻找对应函数名，执行函数后输出回应，回应通过原路返回，将结果显示在控制终端上。

当使用串口连接设备与控制终端时，FinSH 命令的执行流程，如下图所示：

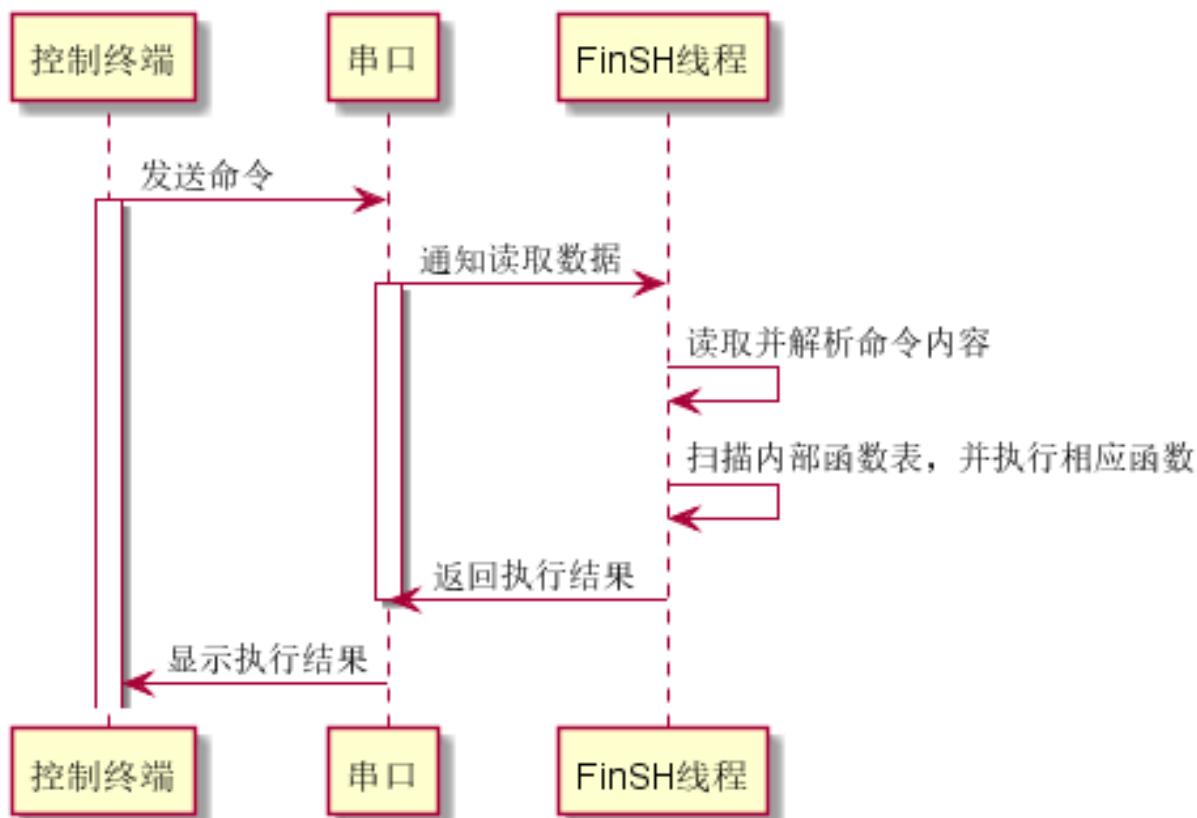


图 22.2: FinSH 命令执行流程图

FinSH 支持权限验证功能，系统在启动后会进行权限验证，只有权限验证通过，才会开启 FinSH 功能，提升系统输入的安全性。

FinSH 支持自动补全、查看历史命令等功能，通过键盘上的按键可以很方便的使用这些功能，FinSH 支持的按键如下表所示：

按键	功能描述
Tab 键	当没有输入任何字符时按下 Tab 键将会打印当前系统支持的所有命令。若已经输入部分字符时按下 Tab 键，将会查找匹配的命令，也会按照文件系统的当前目录下的文件名进行补全，并可以继续输入，多次补全
↑↓ 键	上下翻阅最近输入的历史命令
退格键	删除符
←→ 键	向左或向右移动光标

FinSH 支持两种输入模式，分别是传统命令行模式和 C 语言解释器模式。

22.1.1 传统命令行模式

此模式又称为 msh(module shell)，msh 模式下，FinSH 与传统 shell (dos/bash) 执行方式一致，例如，可以通过 `cd` / 命令将目录切换至根目录。

`msh` 通过解析，将输入字符分解成以空格区分开的命令和参数。其命令执行格式如下所示：

```
command [arg1] [arg2] [...]
```

其中 `command` 既可以是 RT-Thread 内置的命令，也可以是可执行的文件。

22.1.2 C 语言解释器模式

此模式又称为 C-Style 模式。C 语言解释器模式下，FinSH 能够解析执行大部分 C 语言的表达式，并使用类似 C 语言的函数调用方式访问系统中的函数及全局变量，此外它也能够通过命令行方式创建变量。在该模式下，输入的命令必须类似 C 语言中的函数调用方式，即必须携带 () 符号，例如，要输出系统当前所有线程及其状态，在 FinSH 中输入 `list_thread()` 即可打印出需要的信息。FinSH 命令的输出为此函数的返回值。对于一些不存在返回值的函数（`void` 返回值），这个打印输出没有意义。

最初 FinSH 仅支持 C-Style 模式，后来随着 RT-Thread 的不断发展，C-Style 模式在运行脚本或者程序时不太方便，而使用传统的 shell 方式则比较方便。另外，C-Style 模式下，FinSH 占用体积比较大。出于这些考虑，在 RT-Thread 中增加了 msh 模式，msh 模式体积小，使用方便，推荐大家使用 msh 模式。

如果在 RT-Thread 中同时使能了这两种模式，那它们可以动态切换，在 msh 模式下输入 `exit` 后回车，即可切换到 C-Style 模式。在 C-Style 模式输入 `msh()` 后回车，即可进入 msh 模式。两种模式的命令不通用，`msh` 命令无法在 C-Style 模式下使用，反之同理。

22.2 FinSH 内置命令

在 RT-Thread 中默认内置了一些 FinSH 命令，在 FinSH 中输入 `help` 后回车或者直接按下 Tab 键，就可以打印当前系统支持的所有命令。C-Style 和 msh 模式下的内置命令基本一致，这里就以 msh 为例。

`msh` 模式下，按下 Tab 键后可以列出当前支持的所有命令。默认命令的数量不是固定的，RT-Thread 的各个组件会向 FinSH 输出一些命令。例如，当打开 DFS 组件时，就会把 `ls`, `cp`, `cd` 等命令加到 FinSH 中，方便开发者调试。

以下为按下 Tab 键后打印出来的当前支持的所有显示 RT-Thread 内核状态信息的命令，左边是命令名称，右边是关于命令的描述：

<code>RT-Thread shell commands:</code>	
<code>version</code>	- show RT-Thread version information
<code>list_thread</code>	- list thread
<code>list_sem</code>	- list semaphore in system
<code>list_event</code>	- list event in system
<code>list_mutex</code>	- list mutex in system
<code>list_mailbox</code>	- list mail box in system
<code>list_msgqueue</code>	- list message queue in system
<code>list_timer</code>	- list timer in system
<code>list_device</code>	- list device in system
<code>exit</code>	- return to RT-Thread shell mode.

help	- RT-Thread shell help.
ps	- List threads in the system.
time	- Execute command with time.
free	- Show the memory usage in the system.

这里列出输入常用命令后返回的字段信息，方便开发者理解返回的信息内容。

22.2.1 显示线程状态

使用 ps 或者 list_thread 命令来列出系统中的所有线程信息，包括线程优先级、状态、栈的最大使用量等。

msh />list_thread								
thread	pri	status	sp	stack size	max used	left tick	error	
tshell	20	ready	0x00000118	0x00001000	29%	0x00000009	000	
tidle	31	ready	0x0000005c	0x00000200	28%	0x00000005	000	
timer	4	suspend	0x00000078	0x00000400	11%	0x00000009	000	

list_thread 返回字段的描述:

字段	描述
thread	线程的名称
pri	线程的优先级
status	线程当前的状态
sp	线程当前的栈位置
stack size	线程的栈大小
max used	线程历史中使用的最大栈位置
left tick	线程剩余的运行节拍数
error	线程的错误码

22.2.2 显示信号量状态

使用 list_sem 命令来显示系统中所有信号量信息，包括信号量的名称、信号量的值和等待这个信号量的线程数目。

msh />list_sem		
semaphore	v	suspend thread
-----	-----	-----
shrx	000 0	
e0	000 0	

list_sem 返回字段的描述:

字段	描述
semaphore	信号量的名称
v	信号量当前的值
suspend thread	等待这个信号量的线程数目

22.2.3 显示事件状态

使用 `list_event` 命令来显示系统中所有的事件信息，包括事件名称、事件的值和等待这个事件的线程数目。

```
msh />list_event
event      set      suspend thread
-----
```

`list_event` 返回字段的描述：

字段	描述
event	事件集的名称
set	事件集中当前发生的事件
suspend thread	在这个事件集中等待事件的线程数目

22.2.4 显示互斥量状态

使用 `list_mutex` 命令来显示系统中所有的互斥量信息，包括互斥量名称、互斥量的所有者和所有者在互斥量上持有的嵌套次数等。

```
msh />list_mutex
mutex      owner   hold suspend thread
-----
fat0      (NULL)   0000 0
sal_lock (NULL)   0000 0
```

`list_mutex` 返回字段的描述：

字段	描述
mutex	互斥量的名称
owner	当前持有互斥量的线程
hold	持有者在这个互斥量上嵌套持有的次数
suspend thread	等待这个互斥量的线程数目

22.2.5 显示邮箱状态

使用 `list_mailbox` 命令显示系统中所有的邮箱信息，包括邮箱名称、邮箱中邮件的数目和邮箱能容纳邮件的最大数目等。

```
msh />list_mailbox
mailbox entry size suspend thread
-----
etxmb    0000  0008 1:etx
erxmb    0000  0008 1:erx
```

`list_mailbox` 返回字段的描述:

字段	描述
mailbox	邮箱的名称
entry	邮箱中包含的邮件数目
size	邮箱能够容纳的最大邮件数目
suspend thread	等待这个邮箱的线程数目

22.2.6 显示消息队列状态

使用 `list_msgqueue` 命令来显示系统中所有的消息队列信息，包括消息队列的名称、包含的消息数目和等待这个消息队列的线程数目。

```
msh />list_msgqueue
msgqueue entry suspend thread
-----
```

`list_msgqueue` 返回字段的描述:

字段	描述
msgqueue	消息队列的名称
entry	消息队列当前包含的消息数目
suspend thread	等待这个消息队列的线程数目

22.2.7 显示内存池状态

使用 `list_mempool` 命令来显示系统中所有的内存池信息，包括内存池的名称、内存池的大小和最大使用的内存大小等。

```
msh />list_mempool
mempool block total free suspend thread
-----
```

```
signal 0012 0032 0032 0
```

`list_mempool` 返回字段的描述:

字段	描述
mempool	内存池名称
block	内存块大小
total	总内存块
free	空闲内存块
suspend thread	等待这个内存池的线程数目

22.2.8 显示定时器状态

使用 `list_timer` 命令来显示系统中所有的定时器信息，包括定时器的名称、是否是周期性定时器和定时器超时的节拍数等。

```
msh />list_timer
timer      periodic    timeout      flag
-----
tshell    0x00000000 0x00000000 deactivated
tidle     0x00000000 0x00000000 deactivated
timer     0x00000000 0x00000000 deactivated
```

`list_timer` 返回字段的描述:

字段	描述
timer	定时器的名称
periodic	定时器是否是周期性的
timeout	定时器超时时的节拍数
flag	定时器的状态，activated 表示活动的，deactivated 表示不活动的

22.2.9 显示设备状态

使用 `list_device` 命令来显示系统中所有的设备信息，包括设备名称、设备类型和设备被打开次数。

```
msh />list_device
device      type      ref count
-----
e0        Network Interface 0
uart0    Character Device  2
```

`list_device` 返回字段的描述:

字段	描述
device	设备的名称
type	设备的类型
ref count	设备被打开次数

22.2.10 显示动态内存状态

使用 free 命令来显示系统中所有的内存信息。

```
msh />free
total memory: 7669836
used memory : 15240
maximum allocated memory: 18520
```

free 返回字段的描述:

字段	描述
total memory	内存总大小
used memory	已使用的内存大小
maximum allocated memory	最大分配内存

22.3 自定义 FinSH 命令

除了 FinSH 自带的命令，FinSH 还也提供了多个宏接口来导出自定义命令，导出的命令可以直接在 FinSH 中执行。

22.3.1 自定义 msh 命令

自定义的 msh 命令，可以在 msh 模式下被运行，将一个命令导出到 msh 模式可以使用如下宏接口：

```
MSH_CMD_EXPORT(name, desc);
```

参数	描述
name	要导出的命令
desc	导出命令的描述

这个命令可以导出有参数的命令，也可以导出无参数的命令。导出无参数命令时，函数的入参为 void，示例如下：

```
void hello(void)
```

```
{
    rt_kprintf("hello RT-Thread!\n");
}

MSH_CMD_EXPORT(hello , say hello to RT-Thread);
```

导出有参数的命令时，函数的入参为 `int argc` 和 `char**argv`。`argc` 表示参数的个数，`argv` 表示命令行参数字符串指针数组指针。导出有参数命令示例如下：

```
static void atcmd(int argc, char**argv)
{
    .....
}

MSH_CMD_EXPORT(atcmd, atcmd sample: atcmd <server|client>);
```

22.3.2 自定义 C-Style 命令和变量

将自定义命令导出到 C-Style 模式可以使用如下接口：

```
FINSH_FUNCTION_EXPORT(name, desc);
```

参数	描述
<code>name</code>	要导出的命令
<code>desc</code>	导出命令的描述

以下示例定义了一个 `hello` 函数，并将它导出成 C-Style 模式下的命令：

```
void hello(void)
{
    rt_kprintf("hello RT-Thread!\n");
}

FINSH_FUNCTION_EXPORT(hello , say hello to RT-Thread);
```

按照类似的方式，也可以导出一个变量，可以通过如下接口：

```
FINSH_VAR_EXPORT(name, type, desc);
```

参数	描述
<code>name</code>	要导出的变量
<code>type</code>	变量的类型
<code>desc</code>	导出变量的描述

以下示例定义了一个 dummy 变量，并将它导出成 C-Style 模式下的变量命令：

```
static int dummy = 0;
FINSH_VAR_EXPORT(dummy, finsh_type_int, dummy variable for finsh)
```

22.3.3 自定义命令重命名

FinSH 的函数名字长度有一定限制，它由 `finsh.h` 中的宏定义 `FINSH_NAME_MAX` 控制，默认是 16 字节，这意味着 FinSH 命令长度不会超过 16 字节。这里有个潜在的问题：当一个函数名长度超过 `FINSH_NAME_MAX` 时，使用 `FINSH_FUNCTION_EXPORT` 导出这个函数到命令表中后，在 FinSH 符号表中看到完整的函数名，但是完整输入执行会出现 `null node` 错误。这是因为虽然显示了完整的函数名，但是实际上 FinSH 中却保存了前 16 字节作为命令，过多的输入会导致无法正确找到命令，这时就可以使用 `FINSH_FUNCTION_EXPORT_ALIAS` 来对导出的命令进行重命名。

```
FINSH_FUNCTION_EXPORT_ALIAS(name, alias, desc);
```

参数	描述
<code>name</code>	要导出的命令
<code>alias</code>	导出到 FinSH 时显示的名字
<code>desc</code>	导出命令的描述

在重命名的命令名字前加 `_cmd_` 就可以将命令导出到 msh 模式，否则，命令会被导出到 C-Style 模式。以下示例定义了一个 `hello` 函数，并将它重命名为 `ho` 后导出成 C-Style 模式下的命令。

```
void hello(void)
{
    rt_kprintf("hello RT-Thread!\n");
}

FINSH_FUNCTION_EXPORT_ALIAS(hello, ho, say hello to RT-Thread);
```

22.4 FinSH 功能配置

FinSH 功能可以裁剪，宏配置选项在 `rtconfig.h` 文件中定义，具体配置项如下表所示。

宏定义	取值类 型	描述	默认值
<code>#define RT_USING_FINSH</code>	无	使能 FinSH	开启
<code>#define FINSH_THREAD_NAME</code>	字符串	FinSH 线程的名字	“tshell”
<code>#define FINSH_USING_HISTORY</code>	无	打开历史回溯功能	开启
<code>#define FINSH_HISTORY_LINES</code>	整数型	能回溯的历史命令行数	5

宏定义	取值类型	描述	默认值
#define FINSH_USING_SYMTAB	无	可以在 FinSH 中使用符号表	开启
#define FINSH_USING_DESCRIPTION	无	给每个 FinSH 的符号添加一段描述	开启
#define FINSH_USING_MSH	无	使能 msh 模式	开启
#define FINSH_USING_MSH_ONLY	无	只使用 msh 模式	开启
#define FINSH_ARG_MAX	整数型	最大输入参数数量	10
#define FINSH_USING_AUTH	无	使能权限验证	关闭
#define FINSH_DEFAULT_PASSWORD	字符串	权限验证密码	关闭

rtconfig.h 中的参考配置示例如下所示，可以根据实际功能需求情况进行配置。

```
/* 开启 FinSH */
#define RT_USING_FINSH

/* 将线程名称定义为 tshell */
#define FINSH_THREAD_NAME "tshell"

/* 开启历史命令 */
#define FINSH_USING_HISTORY
/* 记录 5 行历史命令 */
#define FINSH_HISTORY_LINES 5

/* 开启使用 Tab 键 */
#define FINSH_USING_SYMTAB
/* 开启描述功能 */
#define FINSH_USING_DESCRIPTION

/* 定义 FinSH 线程优先级为 20 */
#define FINSH_THREAD_PRIORITY 20
/* 定义 FinSH 线程的栈大小为 4KB */
#define FINSH_THREAD_STACK_SIZE 4096
/* 定义命令字符长度为 80 字节 */
#define FINSH_CMD_SIZE 80

/* 开启 msh 功能 */
#define FINSH_USING_MSH
/* 默认使用 msh 功能 */
#define FINSH_USING_MSH_DEFAULT
/* 最大输入参数数量为 10 个 */
#define FINSH_ARG_MAX 10
```

22.5 FinSH 应用示例

22.5.1 不带参数的 msh 命令示例

本小节将演示如何将一个自定义的命令导出到 msh 中，示例代码如下所示，代码中创建了 hello 函数，然后通过 MSH_CMD_EXPORT 命令即可将 hello 函数导出到 FinSH 命令列表中。

```
#include <rtthread.h>

void hello(void)
{
    rt_kprintf("hello RT-Thread!\n");
}

MSH_CMD_EXPORT(hello , say hello to RT-Thread);
```

系统运行起来后，在 FinSH 控制台按 tab 键可以看到导出的命令：

```
msh />
RT-Thread shell commands:
hello          - say hello to RT-Thread
version        - show RT-Thread version information
list_thread    - list thread
.....
```

运行 hello 命令，运行结果如下所示：

```
msh />hello
hello RT_Thread!
msh />
```

22.5.2 带参数的 msh 命令示例

本小节将演示如何将一个带参数的自定义的命令导出到 FinSH 中，示例代码如下所示，代码中创建了 atcmd() 函数，然后通过 MSH_CMD_EXPORT 命令即可将 atcmd() 函数导出到 msh 命令列表中。

```
#include <rtthread.h>

static void atcmd(int argc, char**argv)
{
    if (argc < 2)
    {
        rt_kprintf("Please input'atcmd <server|client>'\n");
        return;
    }

    if (!rt_strcmp(argv[1], "server"))
    {
```

```

        rt_kprintf("AT server!\n");
    }
    else if (!rt_strcmp(argv[1], "client"))
    {
        rt_kprintf("AT client!\n");
    }
    else
    {
        rt_kprintf("Please input'atcmd <server|client>'\n");
    }
}

MSH_CMD_EXPORT(atcmd, atcmd sample: atcmd <server|client>);

```

系统运行起来后，在 FinSH 控制台按 tab 键可以看到导出的命令：

```

msh />
RT-Thread shell commands:
hello          - say hello to RT-Thread
atcmd          - atcmd sample: atcmd <server|client>
version        - show RT-Thread version information
list_thread    - list thread
.....

```

运行 atcmd 命令，运行结果如下所示：

```

msh />atcmd
Please input 'atcmd <server|client>'
msh />

```

运行 atcmd server 命令，运行结果如下所示：

```

msh />atcmd server
AT server!
msh />

```

运行 atcmd client 命令，运行结果如下所示：

```

msh />atcmd client
AT client!
msh />

```

22.6 FinSH 移植

FinSH 完全采用 ANSI C 编写，具备极好的移植性；内存占用少，如果不使用前面章节中介绍的函数方式动态地向 FinSH 添加符号，FinSH 将不会动态申请内存。FinSH 源码位于 `components/finsh` 目录下。移植 FinSH 需要注意以下几个方面：

- FinSH 线程:

每次的命令执行都是在 FinSH 线程（即 tshell 线程）的上下文中完成的。当定义 RT_USING_FINSH 宏时，就可以在初始化线程中调用 `finsh_system_init()` 初始化 FinSH 线程。RT-Thread 1.2.0 之后的版本中可以不使用 `finsh_set_device(const char* device_name)` 函数去显式指定使用的设备，而是会自动调用 `rt_console_get_device()` 函数去使用 `console` 设备（RT-Thread 1.1.x 及以下版本中必须使用 `finsh_set_device(const char* device_name)` 指定 FinSH 使用的设备）。FinSH 线程在函数 `finsh_system_init()` 函数中被创建，它将一直等待 `rx_sem` 信号量。

- FinSH 的输出:

FinSH 的输出依赖于系统的输出，在 RT-Thread 中依赖 `rt_kprintf()` 输出。在启动函数 `rt_hw_board_init()` 中，`rt_console_set_device(const char* name)` 函数设置了 FinSH 的打印输出设备。

- FinSH 的输入:

FinSH 线程在获得了 `rx_sem` 信号量后，调用 `rt_device_read()` 函数从设备（选用串口设备）中获得一个字符然后处理。所以 FinSH 的移植需要 `rt_device_read()` 函数的实现。而 `rx_sem` 信号量的释放通过调用 `rx_indicate()` 函数以完成对 FinSH 线程的输入通知。通常的过程是，当串口接收中断发生时（即串口有输入），接受中断服务例程调用 `rx_indicate()` 函数通知 FinSH 线程有输入，而后 FinSH 线程获取串口输入最后做相应的命令处理。

第 23 章

虚拟文件系统

在早期的嵌入式系统中，需要存储的数据比较少，数据类型也比较单一，往往使用直接在存储设备中的指定地址写入数据的方法来存储数据。然而随着嵌入式设备功能的发展，需要存储的数据越来越多，也越来越复杂，这时仍使用旧方法来存储并管理数据就变得非常繁琐困难。因此我们需要新的数据管理方式来简化存储数据的组织形式，这种方式就是我们接下来要介绍的文件系统。

文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型 (Abstract data type)，是一种用于向用户提供底层数据访问的机制。文件系统通常存储的基本单位是文件，即数据是按照一个个文件的方式进行组织。当文件比较多时，将导致文件繁多，不易分类、重名的问题。而文件夹作为一个容纳多个文件的容器而存在。

本章讲解 RT-Thread 文件系统相关内容，带你了解 RT-Thread 虚拟文件系统的架构、功能特点和使用方式。

23.1 DFS 简介

DFS 是 RT-Thread 提供的虚拟文件系统组件，全称为 Device File System，即设备虚拟文件系统，文件系统的名称使用类似 UNIX 文件、文件夹的风格，目录结构如下图所示：

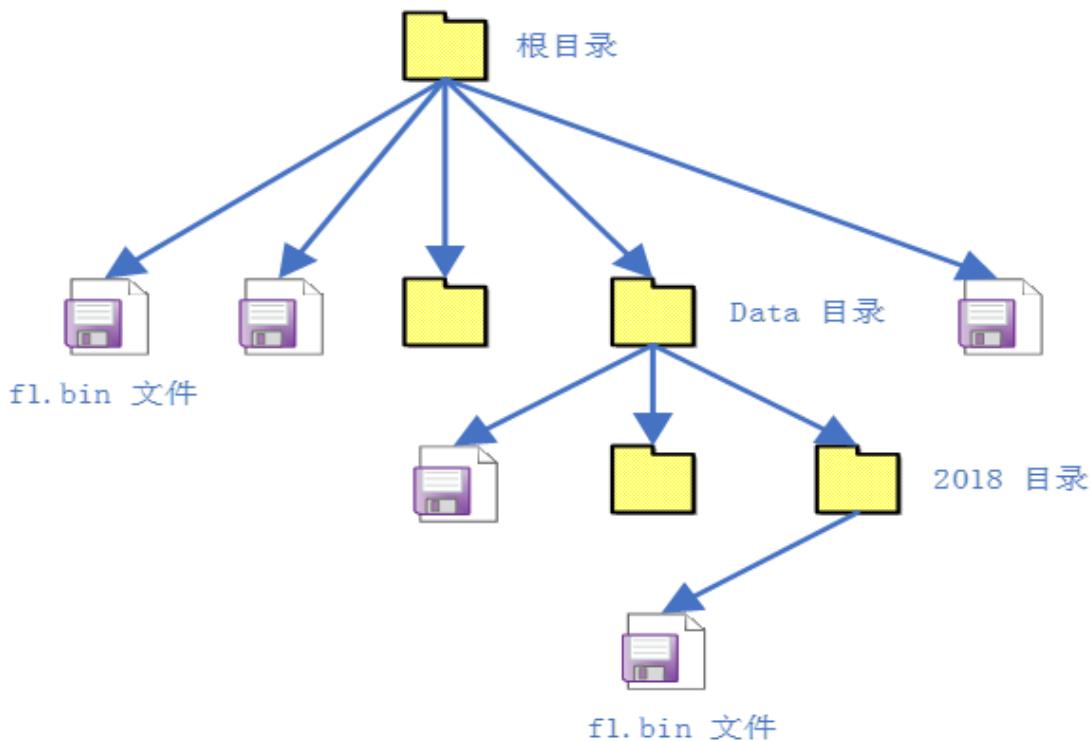


图 23.1: 目录结构图

在 RT-Thread DFS 中，文件系统有统一的根目录，使用 / 来表示。而在根目录下的 f1.bin 文件则使用 /f1.bin 来表示，2018 目录下的 f1.bin 目录则使用 /data/2018/f1.bin 来表示。即目录的分割符号是 /，这与 UNIX/Linux 完全相同，与 Windows 则不相同（Windows 操作系统上使用 \ 来作为目录的分割符）。

23.1.1 DFS 架构

RT-Thread DFS 组件的主要功能特点有：

- 为应用程序提供统一的 POSIX 文件和目录操作接口：read、write、poll/select 等。
- 支持多种类型的文件系统，如 FatFS、RomFS、DevFS 等，并提供普通文件、设备文件、网络文件描述符的管理。
- 支持多种类型的存储设备，如 SD Card、SPI Flash、Nand Flash 等。

DFS 的层次架构如下图所示，主要分为 POSIX 接口层、虚拟文件系统层和设备抽象层。

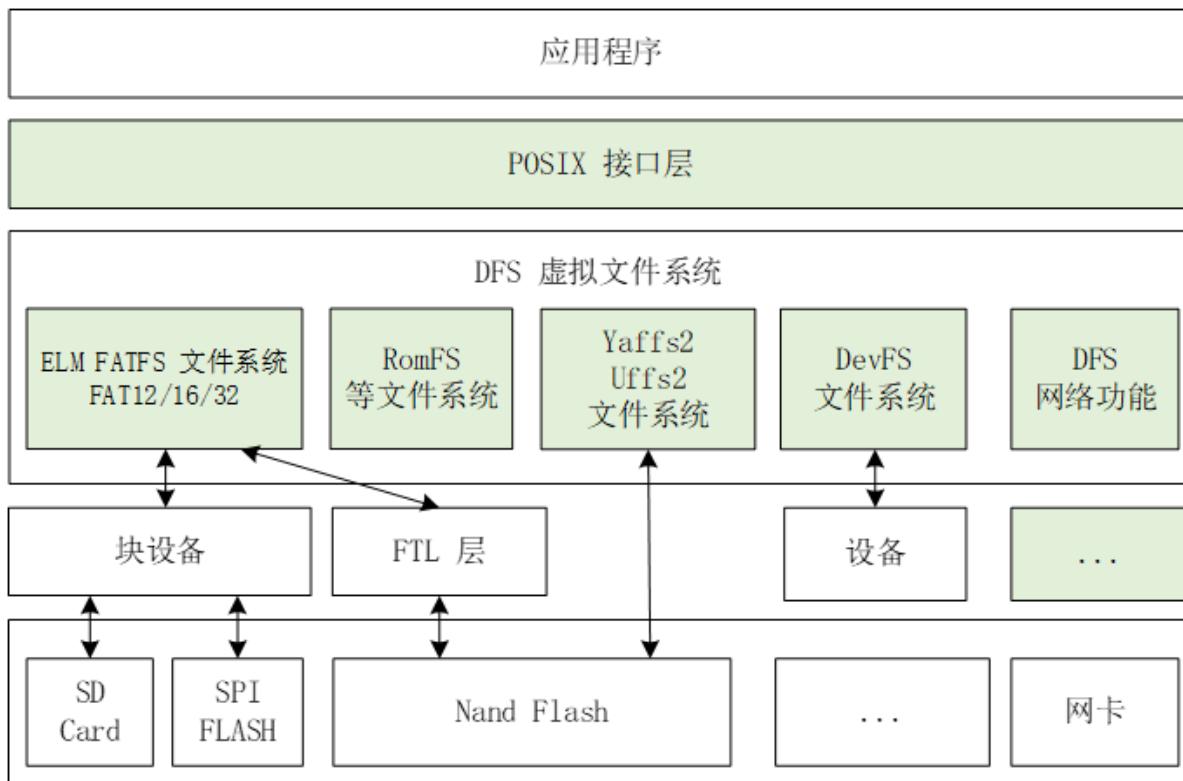


图 23.2: DFS 层次架构图

23.1.2 POSIX 接口层

POSIX 表示可移植操作系统接口（Portable Operating System Interface of UNIX，缩写 POSIX），POSIX 标准定义了操作系统应该为应用程序提供的接口标准，是 IEEE 为要在各种 UNIX 操作系统上运行的软件而定义的一系列 API 标准的总称。

POSIX 标准意在期望获得源代码级别的软件可移植性。换句话说，为一个 POSIX 兼容的操作系统编写的程序，应该可以在任何其它 POSIX 操作系统（即使是来自另一个厂商）上编译执行。RT-Thread 支持 POSIX 标准接口，因此可以很方便的将 Linux/Unix 的程序移植到 RT-Thread 操作系统上。

在类 Unix 系统中，普通文件、设备文件、网络文件描述符是同一种文件描述符。而在 RT-Thread 操作系统中，使用 DFS 来实现这种统一性。有了这种文件描述符的统一性，我们就可以使用 poll/select 接口来对这几种描述符进行统一轮询，为实现程序功能带来方便。

使用 poll/select 接口可以阻塞地同时探测一组支持非阻塞的 I/O 设备是否有事件发生（如可读，可写，有高优先级的错误输出，出现错误等等），直至某一个设备触发了事件或者超过了指定的等待时间。这种机制可以帮助调用者寻找当前就绪的设备，降低编程的复杂度。

23.1.3 虚拟文件系统层

用户可以将具体的文件系统注册到 DFS 中，如 FatFS、RomFS、DevFS 等，下面介绍几种常用的文件系统类型：

- FatFS 是专为小型嵌入式设备开发的一个兼容微软 FAT 格式的文件系统，采用 ANSI C 编写，具有良好的硬件无关性以及可移植性，是 RT-Thread 中最常用的文件系统类型。

- 传统型的 RomFS 文件系统是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存，按顺序存放数据，因而支持应用程序以 XIP(execute In Place, 片内运行) 方式运行，在系统运行时，节省 RAM 空间。
- Jffs2 文件系统是一种日志闪存文件系统。主要用于 NOR 型闪存，基于 MTD 驱动层，特点是：可读写的、支持数据压缩的、基于哈希表的日志型文件系统，并提供了崩溃 / 掉电安全保护，提供写平衡支持等。
- DevFS 即设备文件系统，在 RT-Thread 操作系统中开启该功能后，可以将系统中的设备在 /dev 文件夹下虚拟成文件，使得设备可以按照文件的操作方式使用 read、write 等接口进行操作。
- NFS 网络文件系统（Network File System）是一项在不同机器、不同操作系统之间通过网络共享文件的技术。在操作系统的开发调试阶段，可以利用该技术在主机上建立基于 NFS 的根文件系统，挂载到嵌入式设备上，可以很方便地修改根文件系统的内容。
- UFFS 是 Ultra-low-cost Flash File System（超低功耗的闪存文件系统）的简称。它是国人开发的、专为嵌入式设备等小内存环境中使用 Nand Flash 的开源文件系统。与嵌入式中常使用的 Yaffs 文件系统相比具有资源占用少、启动速度快、免费等优势。

23.1.4 设备抽象层

设备抽象层将物理设备如 SD Card、SPI Flash、Nand Flash，抽象成符合文件系统能够访问的设备，例如 FAT 文件系统要求存储设备必须是块设备类型。

不同文件系统类型是独立于存储设备驱动而实现的，因此把底层存储设备的驱动接口和文件系统对接起来之后，才可以正确地使用文件系统功能。

23.2 挂载管理

文件系统的初始化过程一般分为以下几个步骤：

1. 初始化 DFS 组件。
2. 初始化具体类型的文件系统。
3. 在存储器上创建块设备。
4. 格式化块设备。
5. 挂载块设备到 DFS 目录中。
6. 当文件系统不再使用，可以将它卸载。

23.2.1 初始化 DFS 组件

DFS 组件的初始化是由 `dfs_init()` 函数完成。`dfs_init()` 函数会初始化 DFS 所需的相关资源，创建一些关键的数据结构，有了这些数据结构，DFS 便能在系统中找到特定的文件系统，并获得对特定存储设备内文件的操作方法。如果开启了自动初始化（默认开启），该函数将被自动调用。

23.2.2 注册文件系统

在 DFS 组件初始化之后，还需要初始化使用的具体类型的文件系统，也就是将具体类型的文件系统注册到 DFS 中。注册文件系统的接口如下所示：

```
int dfs_register(const struct dfs_filesystem_ops *ops);
```

参数	描述
ops	文件系统的操作函数的集合
返回	—
0	文件注册成功
-1	文件注册失败

该函数不需要用户调用，他会被不同文件系统的初始化函数调用，如 elm-FAT 文件系统的初始化函数 `elm_init()`。开启对应的文件系统后，如果开启了自动初始化（默认开启），文件系统初始化函数也将被自动调用。

`elm_init()` 函数会初始化 elm-FAT 文件系统，此函数会调用 `dfs_register()` 函数将 elm-FAT 文件系统注册到 DFS 中，文件系统注册过程如下图所示：

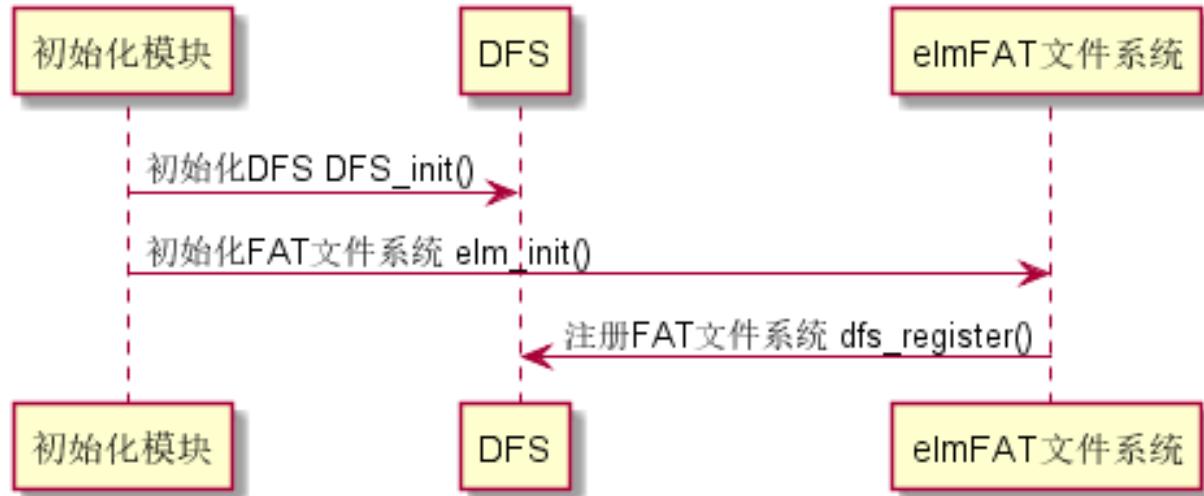


图 23.3: 注册文件系统

23.2.3 将存储设备注册为块设备

因为只有块设备才可以挂载到文件系统上，因此需要在存储设备上创建所需的块设备。如果存储设备是 SPI Flash，则可以使用“串行 Flash 通用驱动库 SFUD”组件，它提供了各种 SPI Flash 的驱动，并将 SPI Flash 抽象成块设备用于挂载，注册块设备过程如下图所示：

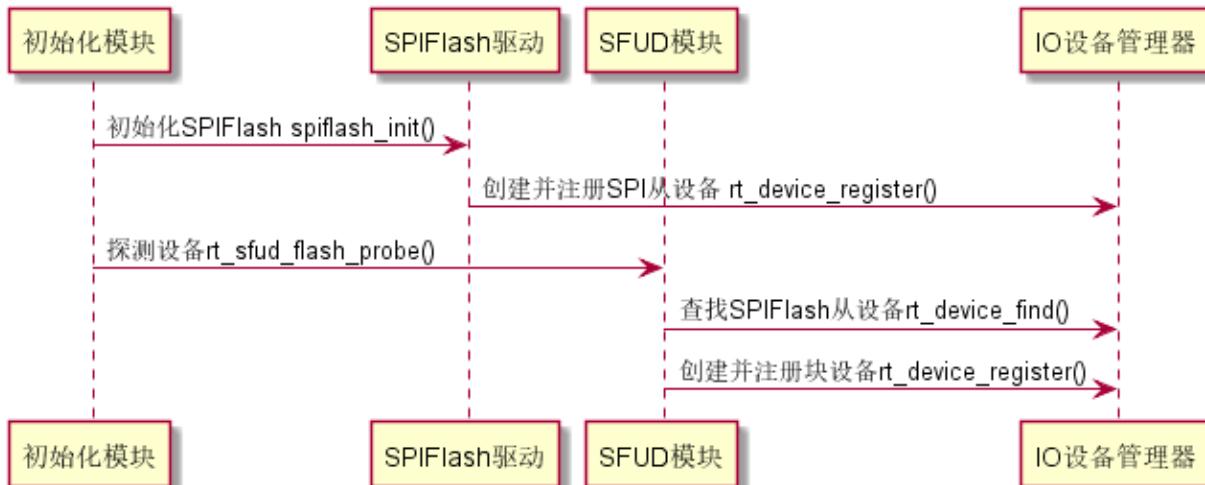


图 23.4: 注册块设备时序图

23.2.4 格式化文件系统

注册了块设备之后，还需要在块设备上创建指定类型的文件系统，也就是格式化文件系统。可以使用 `dfs_mkfs()` 函数对指定的存储设备进行格式化，创建文件系统，格式化文件系统的接口如下所示：

```
int dfs_mkfs(const char * fs_name, const char * device_name);
```

参数	描述
<code>fs_name</code>	文件系统类型
<code>device_name</code>	块设备名称
返回	—
0	文件系统格式化成功
-1	文件系统格式化失败

文件系统类型（`fs_name`）可取值及对应的文件系统如下表所示：

取值	文件系统类型
elm	elm-FAT 文件系统
jffs2	jffs2 日志闪存文件系统
nfs	NFS 网络文件系统
ram	RamFS 文件系统
rom	RomFS 只读文件系统
uffs	uffs 文件系统

以 elm-FAT 文件系统格式化块设备为例，格式化过程如下图所示：

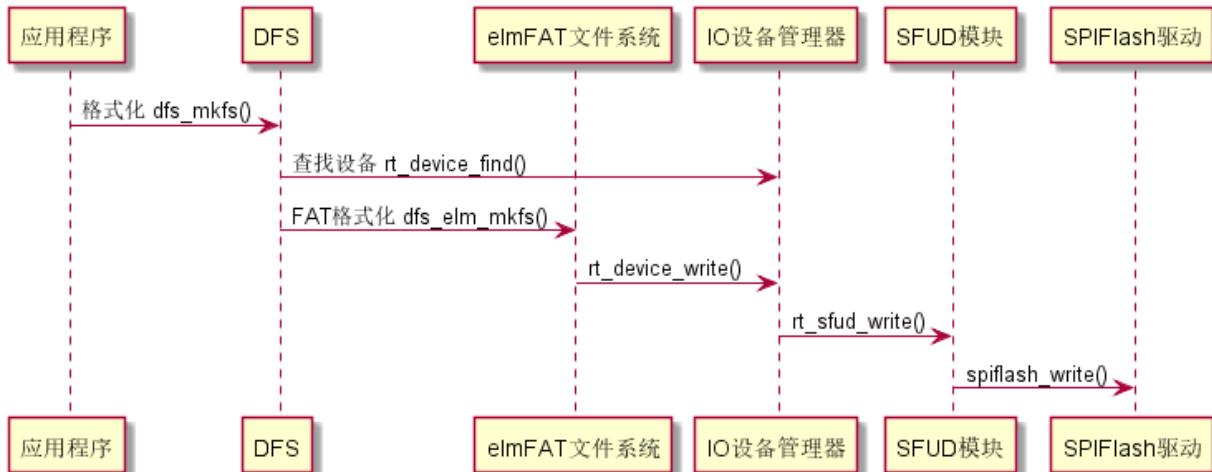


图 23.5: 格式化文件系统

还可以使用 `mkfs` 命令格式化文件系统，格式化块设备 `sd0` 的运行结果如下所示：

```
msh />mkfs sd0          # sd0 为块设备名称，该命令会默认格式化 sd0 为 elm-
      FAT 文件系统
msh />
msh />mkfs -t elm sd0    # 使用 -t 参数指定文件系统类型为 elm-FAT 文件系统
```

23.2.5 挂载文件系统

在 RT-Thread 中，挂载是指将一个存储设备挂接到一个已存在的路径上。我们要访问存储设备中的文件，必须将文件所在的分区挂载到一个已存在的路径上，然后通过这个路径来访问存储设备。挂载文件系统的接口如下所示：

```
int dfs_mount(const char *device_name,
              const char *path,
              const char *filesystemtype,
              unsigned long rwflag,
              const void *data);
```

参数	描述
<code>device_name</code>	已经格式化的块设备名称
<code>path</code>	挂载路径，即挂载点
<code>filesystemtype</code>	挂载的文件系统类型，可取值见 <code>dfs_mkfs()</code> 函数描述
<code>rwflag</code>	读写标志位
<code>data</code>	特定文件系统的私有数据
返回	—
0	文件系统挂载成功
-1	文件系统挂载失败

如果只有一个存储设备，则可以直接挂载到根目录 / 上。

23.2.6 卸载文件系统

当某个文件系统不需要再使用了，那么可以将它卸载掉。卸载文件系统的接口如下所示：

```
int dfs_unmount(const char *specialfile);
```

参数	描述
specialfile	挂载路径
返回	—
0	卸载文件系统成功
-1	卸载文件系统失败

23.3 文件管理

本节介绍对文件进行操作的相关函数，对文件的操作一般都要基于文件描述符 fd，如下图所示：

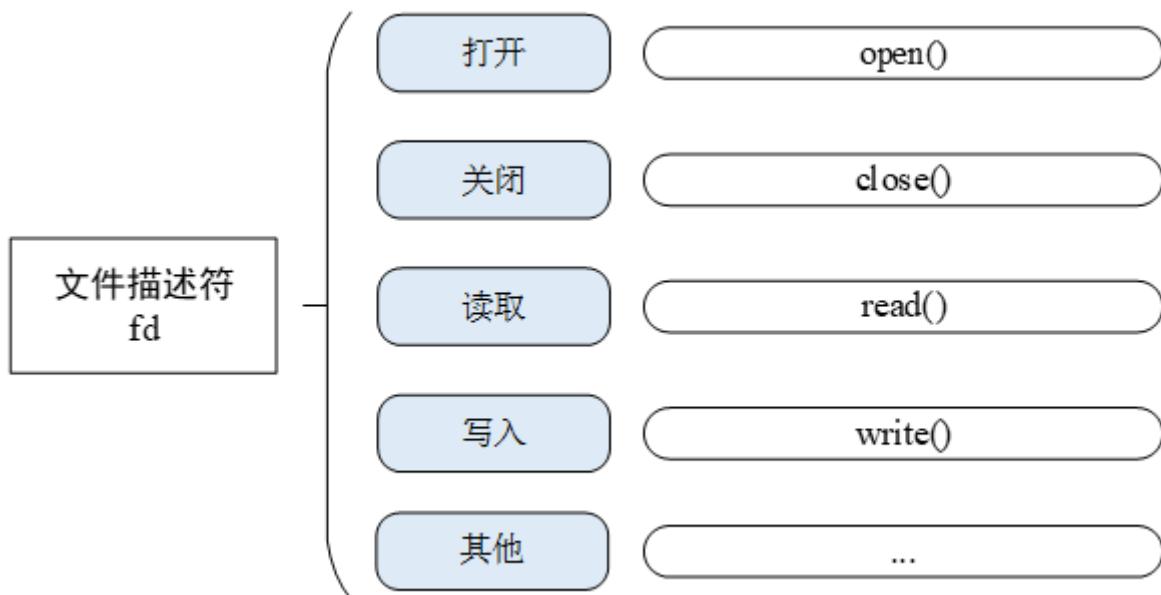


图 23.6: 文件管理常用函数

23.3.1 打开和关闭文件

打开或创建一个文件可以调用下面的 open() 函数：

```
int open(const char *file, int flags, ...);
```

参数	描述
file	打开或创建的文件名
flags	指定打开文件的方式, 取值可参考下表
返回	—
文件描述符	文件打开成功
-1	文件打开失败

一个文件可以以多种方式打开, 并且可以同时指定多种打开方式。例如, 一个文件以 O_RDONLY 和 O_CREAT 的方式打开, 那么当指定打开的文件不存在时, 就会先创建这个文件, 然后再以只读的方式打开。文件打开方式如下表所示:

参数	描述
O_RDONLY	只读方式打开文件
O_WRONLY	只写方式打开文件
O_RDWR	以读写方式打开文件
O_CREAT	如果要打开的文件不存在, 则建立该文件
O_APPEND	当读写文件时会从文件尾开始移动, 也就是所写入的数据会以附加的方式添加到文件的尾部
O_TRUNC	如果文件已经存在, 则清空文件中的内容

当使用完文件后若不再需要使用则可使用 `close()` 函数关闭该文件, 而 `close()` 会让数据写回磁盘, 并释放该文件所占用的资源。

```
int close(int fd);
```

参数	描述
fd	文件描述符
返回	—
0	文件关闭成功
-1	文件关闭失败

23.3.2 读写数据

读取文件内容可使用 `read()` 函数:

```
int read(int fd, void *buf, size_t len);
```

参数	描述
fd	文件描述符
buf	缓冲区指针
len	读取文件的字节数
返回	—
int	实际读取到的字节数
0	读取数据已到达文件结尾或者无可读取的数据
-1	读取出错，错误代码查看当前线程的 <code>errno</code>

该函数会把参数 `fd` 所指的文件的 `len` 个字节读取到 `buf` 指针所指的内存中。此外文件的读写位置指针会随读取到的字节移动。

向文件中写入数据可使用 `write()` 函数：

```
int write(int fd, const void *buf, size_t len);
```

参数	描述
Fd	文件描述符
buf	缓冲区指针
len	写入文件的字节数
返回	—
int	实际写入的字节数
-1	写入出错，错误代码查看当前线程的 <code>errno</code>

该函数会把 `buf` 指针所指向的内存中 `len` 个字节写入到参数 `fd` 所指的文件内。此外文件的读写位置指针会随着写入的字节移动。

23.3.3 重命名

重命名文件可使用 `rename()` 函数：

```
int rename(const char *old, const char *new);
```

参数	描述
old	旧文件名
new	新文件名
返回	—

参数	描述
0	更改名称成功
-1	更改名称失败

该函数会将参数 `old` 所指定的文件名称改为参数 `new` 所指的文件名称。若 `new` 所指定的文件已经存在，则该文件将会被覆盖。

23.3.4 取得状态

获取文件状态可使用下面的 `stat()` 函数：

```
int stat(const char *file, struct stat *buf);
```

参数	描述
<code>file</code>	文件名
<code>buf</code>	结构指针，指向一个存放文件状态信息的结构体
返回	—
0	获取状态成功
-1	获取状态失败

23.3.5 删除文件

删除指定目录下的文件可使用 `unlink()` 函数：

```
int unlink(const char *pathname);
```

参数	描述
<code>pathname</code>	指定删除文件的绝对路径
返回	—
0	删除文件成功
-1	删除文件失败

23.3.6 同步文件数据到存储设备

同步内存中所有已修改的文件数据到储存设备可使用 `fsync()` 函数：

```
int fsync(int fildes);
```

参数	描述
fildes	文件描述符
返回	—
0	同步文件成功
-1	同步文件失败

23.3.7 查询文件系统相关信息

查询文件系统相关信息可使用 `statfs()` 函数：

```
int statfs(const char *path, struct statfs *buf);
```

参数	描述
path	文件系统的挂载路径
buf	用于储存文件系统信息的结构体指针
返回	—
0	查询文件系统信息成功
-1	查询文件系统信息失败

23.3.8 监视 I/O 设备状态

监视 I/O 设备是否有事件发生可使用 `select()` 函数：

```
int select( int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

参数	描述
nfds	集合中所有文件描述符的范围，即所有文件描述符的最大值加 1
readfds	需要监视读变化的文件描述符集合
writefds	需要监视写变化的文件描述符集合
exceptfds	需要监视出现异常的文件描述符集合
timeout	<code>select</code> 的超时时间
返回	—
正值	监视的文件集合出现可读写事件或出错

参数	描述
0	等待超时，没有可读写或错误的文件
负值	出错

使用 `select()` 接口可以阻塞地同时探测一组支持非阻塞的 I/O 设备是否有事件发生（如可读，可写，有高优先级的错误输出，出现错误等等），直至某一个设备触发了事件或者超过了指定的等待时间。

23.4 目录管理

本节介绍目录管理经常使用的函数，对目录的操作一般都基于目录地址，如下图所示：

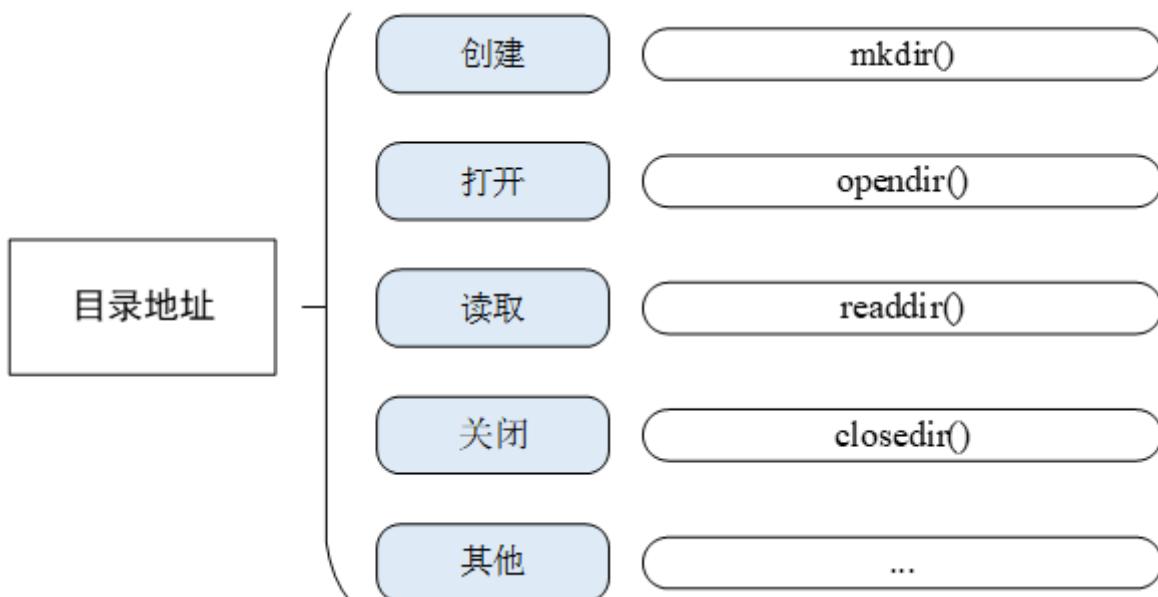


图 23.7：目录管理常用函数

23.4.1 创建和删除目录

创建目录可使用 `mkdir()` 函数：

```
int mkdir(const char *path, mode_t mode);
```

参数	描述
path	目录的绝对地址
mode	创建模式
返回	—
0	创建目录成功
-1	创建目录失败

该函数用来创建一个目录即文件夹，参数 `path` 为目录的绝对路径，参数 `mode` 在当前版本未启用，所以填入默认参数 `0x777` 即可。

删除目录可使用 `rmdir()` 函数：

```
int rmdir(const char *pathname);
```

参数	描述
<code>pathname</code>	需要删除目录的绝对路径
返回	—
0	目录删除成功
-1	目录删除错误

23.4.2 打开和关闭目录

打开目录可使用 `opendir()` 函数：

```
DIR* opendir(const char* name);
```

参数	描述
<code>name</code>	目录的绝对地址
返回	—
<code>DIR</code>	打开目录成功，返回指向目录流的指针
<code>NULL</code>	打开失败

关闭目录可使用 `closedir()` 函数：

```
int closedir(DIR* d);
```

参数	描述
<code>d</code>	目录流指针
返回	—
0	目录关闭成功
-1	目录关闭错误

该函数用来关闭一个目录，必须和 `opendir()` 函数配合使用。

23.4.3 读取目录

读取目录可使用 `readdir()` 函数:

```
struct dirent* readdir(DIR *d);
```

参数	描述
d	目录流指针
返回	—
dirent	读取成功返回指向目录条目的结构体指针
NULL	已读到目录尾

该函数用来读取目录，参数 `d` 为目录流指针。此外，每读取一次目录，目录流的指针位置将自动往后递推 1 个位置。

23.4.4 取得目录流的读取位置

获取目录流的读取位置可使用 `telldir()` 函数:

```
long telldir(DIR *d);
```

参数	描述
d	目录流指针
返回	—
long	读取位置的偏移量

该函数的返回值记录着一个目录流的当前位置，此返回值代表距离目录文件开头的 **偏移量**。你可以在随后的 [seekdir\(\)函数调用](#) 中利用这个值来重置目录到当前位置。也就是说 `telldir()` 函数可以和 `seekdir()` 函数配合使用，重新设置目录流的读取位置到指定的偏移量。

23.4.5 设置下次读取目录的位置

设置下次读取目录的位置可使用 `seekdir()` 函数:

```
void seekdir(DIR *d, off_t offset);
```

参数	描述
d	目录流指针
offset	偏移值，距离本次目录的位移

该用来设置参数 `d` 目录流的读取位置，在调用 `readdir()` 时便从此新位置开始读取。

23.4.6 重设读取目录的位置为开头位置

重设目录流的读取位置为开头可使用 `rewinddir()` 函数：

```
void rewinddir(DIR *d);
```

参数	描述
<code>d</code>	目录流指针

该函数可以用来设置 `d` 目录流目前的读取位置为目录流的初始位置。

23.5 DFS 配置选项

文件系统在 `menuconfig` 中具体配置路径如下：

```
RT-Thread Components --->
    Device virtual file system --->
```

配置菜单描述及对应的宏定义如下表所示：

配置选项	对应宏定义	描述
[*] Using device virtual file system	RT_USING_DFS	开启 DFS 虚拟文件系统
[*] Using working directory	DFS_USING_WORKDIR	开启相对路径
(2) The maximal number of mounted file system	DFS_FILESYSTEMS_MAX	最大挂载文件系统的数量
(2) The maximal number of file system type	DFS_FILESYSTEM_TYPES_MAX	最大支持文件系统的数量
(4) The maximal number of opened files	DFS_FD_MAX	打开文件的最大数量
[] Using mount table for file system	RT_USING_DFS_MNTTABLE	开启自动挂载表
[*] Enable elm-chan fatfs	RT_USING_DFS_ELMFAT	开启 elm-FatFs 文件系统
[*] Using devfs for device objects	RT_USING_DFS_DEVFS	开启 DevFS 设备文件系统
[] Enable ReadOnly file system on flash	RT_USING_DFS_ROMFS	开启 RomFS 文件系统
[] Enable RAM file system	RT_USING_DFS_RAMFS	开启 RamFS 文件系统

配置选项	对应宏定义	描述
[] Enable UFFS file system: Ultra-low-cost Flash File System	RT_USING_DFS_UFFS	开启 UFFS 文件系统
[] Enable JFFS2 file system	RT_USING_DFS_JFFS2	开启 JFFS2 文件系统
[] Using NFS v3 client file system	RT_USING_DFS_NFS	开启 NFS 文件系统

默认情况下，RT-Thread 操作系统为了获得较小的内存占用，并不会开启相对路径功能。当支持相对路径选项没有打开时，在使用文件、目录接口进行操作时应该使用绝对目录进行（因为此时系统中不存在当前工作的目录）。如果需要使用当前工作目录以及相对目录，可在文件系统的配置项中开启相对路径功能。

选项 [*] Using mount table for file system 被选中之后，会使能相应的宏 RT_USING_DFS_MNTTABLE，开启自动挂载表功能。自动挂载表 `mount_table[]` 由用户在应用代码中提供，用户需在表中指定设备名称、挂载路径、文件系统类型、读写标志及私有数据等，之后系统便会遍历该挂载表执行挂载，需要注意的是挂载表必须以 {0} 结尾，用于判断表格结束。

自动挂载表 `mount_table[]` 的示例如下所示，其中 `mount_table[0]` 的 5 个成员即函数 `dfs_mount()` 的 5 个参数，意思是将 elm 文件系统挂载到 flash0 设备的 / 路径下，rwflag 为 0，data 为 0；`mount_table[1]` 为 {0} 作为结尾，用于判断表格结束。

```
const struct dfs_mount_tbl mount_table[] =
{
    {"flash0", "/", "elm", 0, 0},
    {0}
};
```

23.5.1 elm-FatFs 文件系统配置选项

在 menuconfig 中开启 elm-FatFs 文件系统后可对 elm-FatFs 做进一步配置，配置菜单描述及对应的宏定义如下表所示：

配置选项	对应宏定义	描述
(437) OEM code page	RT_DFS_ELM_CODE_PAGE	编码方式
[*] Using RT_DFS_ELM_WORD_ACCESS	RT_DFS_ELM_WORD_ACCESS	
Support long file name (0: LFN disable) —>	RT_DFS_ELM_USE_LFN	开启长文件名子菜单
(255) Maximal size of file name length	RT_DFS_ELM_MAX_LFN	文件名最大长度
(2) Number of volumes (logical drives) to be used.	RT_DFS_ELM_DRIVES	挂载 FatFs 的设备数 量
(4096) Maximum sector size to be handled	RT_DFS_ELM_MAX_SECTOR_SIZE	文件系统扇区大小

配置选项	对应宏定义	描述
[] Enable sector erase feature	RT_DFS_ELM_USE_ERASE	
[*] Enable the reentrancy (thread safe) of the FatFs module	RT_DFS_ELM_REENTRANT	开启可重入性

23.5.1.1 长文件名

默认情况下，FatFs 的文件命名有如下缺点：

- 文件名（不含后缀）最长不超过 8 个字符，后缀最长不超过 3 个字符，文件名和后缀超过限制后将会被截断。
- 文件名不支持大小写（显示为大写）

如果需要支持长文件名，则需要打开支持长文件名选项。长文件名子菜单描述如下所示：

配置选项	对应宏定义	描述
() 0: LFN disable	RT_DFS_ELM_USE_LFN_0	关闭长文件名
() 1: LFN with static LFN working buffer	RT_DFS_ELM_USE_LFN_1	采用静态缓冲区支持长文件名，多线程操作文件名时将会带来重入问题
() 2: LFN with dynamic LFN working buffer on the stack	RT_DFS_ELM_USE_LFN_2	采用栈内临时缓冲区支持长文件名。对栈空间需求较大
(X) 3: LFN with dynamic LFN working buffer on the heap	RT_DFS_ELM_USE_LFN_3	使用 heap (malloc 申请) 缓冲区存放长文件名，最安全（默认方式）

23.5.1.2 编码方式

当打开长文件名支持时，可以设置文件名的编码方式，RT-Thread/FatFs 默认使用 437 编码（美国英语）。如果需要存储中文文件名，可以使用 936 编码（GBK 编码）。936 编码需要一个大约 180KB 的字库。如果仅使用英文字符作为文件，建议使用 437 编码（美国英语），这样就可以节省这 180KB 的 Flash 空间。

FatFs 所支持的文件编码如下所示：

```
/* This option specifies the OEM code page to be used on the target system.
/ Incorrect setting of the code page can cause a file open failure.
/
/   1 - ASCII (No extended character. Non-LFN cfg. only)
/   437 - U.S.
/   720 - Arabic
/   737 - Greek
```

```

/ 771 - KBL
/ 775 - Baltic
/ 850 - Latin 1
/ 852 - Latin 2
/ 855 - Cyrillic
/ 857 - Turkish
/ 860 - Portuguese
/ 861 - Icelandic
/ 862 - Hebrew
/ 863 - Canadian French
/ 864 - Arabic
/ 865 - Nordic
/ 866 - Russian
/ 869 - Greek 2
/ 932 - Japanese (DBCS)
/ 936 - Simplified Chinese (DBCS)
/ 949 - Korean (DBCS)
/ 950 - Traditional Chinese (DBCS)
*/

```

23.5.1.3 文件系统扇区大小

指定 FatFs 的内部扇区大小，需要大于等于实际硬件驱动的扇区大小。例如，某 spi flash 芯片扇区为 4096 字节，则上述宏需要修改为 4096，否则 FatFs 从驱动读入数据时就会发生数组越界而导致系统崩溃（新版本在系统执行时给出警告信息）。

一般 Flash 设备可以设置为 4096，常见的 TF 卡和 SD 卡的扇区大小设置为 512。

23.5.1.4 可重入性

FatFs 充分考虑了多线程安全读写安全的情况，当在多线程中读写 FatFs 时，为了避免重入带来的问题，需要打开上述宏。如果系统仅有一个线程操作文件系统，不会出现重入问题，则可以关闭此功能节省资源。

23.5.1.5 更多配置

FatFs 本身支持非常多的配置选项，配置非常灵活。下面文件为 FatFs 的配置文件，可以修改这个文件来定制 FatFs。

```
components/dfs/filesystems/elmfat/ffconf.h
```

23.6 DFS 应用示例

23.6.1 FinSH 命令

文件系统挂载成功后就可以进行文件和目录的操作了，文件系统操作常用的 FinSH 命令如下表所示：

FinSH 命令	描述
ls	显示文件和目录的信息
cd	进入指定目录
cp	复制文件
rm	删除文件或目录
mv	将文件移动位置或改名
echo	将指定内容写入指定文件，当文件存在时，就写入该文件，当文件不存在时就新创建一个文件并写入
cat	展示文件的内容
pwd	打印出当前目录地址
mkdir	创建文件夹
mkfs	格式化文件系统

使用 `ls` 命令查看当前目录信息，运行结果如下所示：

```
msh />ls          # 使用 ls 命令查看文件系统目录信息
Directory /:      # 可以看到已经存在根目录 /
```

使用 `mkdir` 命令来创建文件夹，运行结果如下所示：

```
msh />mkdir rt-thread      # 创建 rt-thread 文件夹
msh />ls                  # 查看目录信息如下
Directory /:
rt-thread                <DIR>
```

使用 `echo` 命令将输入的字符串输出到指定输出位置，运行结果如下所示：

```
msh />echo "hello rt-thread!!!"
          # 将字符串输出到标准输出
hello rt-thread!!!
msh />echo "hello rt-thread!!!" hello.txt      # 将字符串输出到 hello.txt 文件
msh />ls
Directory /:
rt-thread                <DIR>
hello.txt                 18
msh />
```

使用 `cat` 命令查看文件内容，运行结果如下所示：

```
msh />cat hello.txt      # 查看 hello.txt 文件的内容并输出
hello rt-thread!!!
```

使用 `rm` 命令删除文件夹或文件，运行结果如下所示：

```
msh />ls          # 查看当前目录信息
```

```

Directory /:
rt-thread      <DIR>
hello.txt      18
msh />rm rt-thread          # 删除 rt-thread 文件夹
msh />ls
Directory /:
hello.txt      18
msh />rm hello.txt          # 删除 hello.txt 文件
msh />ls
Directory /:
msh />

```

23.6.2 读写文件示例

文件系统正常工作后，就可以运行应用示例，在该示例代码中，首先会使用 `open()` 函数创建一个文件 `text.txt`，并使用 `write()` 函数在文件中写入字符串 “`RT-Thread Programmer!\n`”，然后关闭文件。再次使用 `open()` 函数打开 `text.txt` 文件，读出其中的内容并打印出来，最后关闭该文件。

示例代码如下所示：

```

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

static void readwrite_sample(void)
{
    int fd, size;
    char s[] = "RT-Thread Programmer!", buffer[80];

    rt_kprintf("Write string %s to test.txt.\n", s);

    /* 以创建和读写模式打开 /text.txt 文件，如果该文件不存在则创建该文件 */
    fd = open("/text.txt", O_WRONLY | O_CREAT);
    if (fd>= 0)
    {
        write(fd, s, sizeof(s));
        close(fd);
        rt_kprintf("Write done.\n");
    }

    /* 以只读模式打开 /text.txt 文件 */
    fd = open("/text.txt", O_RDONLY);
    if (fd>= 0)
    {
        size = read(fd, buffer, sizeof(buffer));
        close(fd);
        rt_kprintf("Read from file test.txt : %s \n", buffer);
        if (size < 0)
            return ;
    }
}

```

```

    }
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(readwrite_sample, readwrite sample);

```

23.6.3 更改文件名称示例

本小节的示例代码展示如何修改文件名称，程序会创建一个操作文件的函数 `rename_sample()` 并导出到 `msh` 命令列表。该函数会调用 `rename()` 函数，将名为 `text.txt` 的文件改名为 `text1.txt`。示例代码如下所示：

```

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

static void rename_sample(void)
{
    rt_kprintf("%s => %s", "/text.txt", "/text1.txt");

    if (rename("/text.txt", "/text1.txt") < 0)
        rt_kprintf("[error!]\n");
    else
        rt_kprintf("[ok!]\n");
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(rename_sample, rename sample);

```

在 FinSH 控制台运行该示例，运行结果如下：

```

msh />echo "hello" text.txt
msh />ls
Directory /:
text.txt      5
msh />rename_sample
/text.txt => /text1.txt [ok!]
msh />ls
Directory /:
text1.txt      5

```

在示例展示过程中，我们先使用 `echo` 命令创建一个名为 `text.txt` 文件，然后运行示例代码将文件 `text.txt` 的文件名修改为 `text1.txt`。

23.6.4 获取文件状态示例

本小节的示例代码展示如何获取文件状态，程序会创建一个操作文件的函数 `stat_sample()` 并导出到 `msh` 命令列表。该函数会调用 `stat()` 函数获取 `text.txt` 文件的文件大小信息。示例代码如下所示：

```

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

```

```

static void stat_sample(void)
{
    int ret;
    struct stat buf;
    ret = stat("/text.txt", &buf);
    if(ret == 0)
        rt_kprintf("text.txt file size = %d\n", buf.st_size);
    else
        rt_kprintf("text.txt file not found\n");
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(stat_sample, show text.txt stat sample);

```

在 FinSH 控制台运行该示例，运行结果如下：

```

msh />echo "hello" text.txt
msh />stat_sample
text.txt file size = 5

```

在示例运行过程中，首先会使用 `echo` 命令创建文件 `text.txt`，然后运行示例代码，将文件 `text.txt` 的文件大小信息打印出来。

23.6.5 创建目录示例

本小节的示例代码展示如何创建目录，程序会创建一个操作文件的函数 `mkdir_sample()` 并导出到 `msh` 命令列表，该函数会调用 `mkdir()` 函数创建一个名为 `dir_test` 的文件夹。示例代码如下所示：

```

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

static void mkdir_sample(void)
{
    int ret;

    /* 创建目录 */
    ret = mkdir("/dir_test", 0x777);
    if (ret < 0)
    {
        /* 创建目录失败 */
        rt_kprintf("dir error!\n");
    }
    else
    {
        /* 创建目录成功 */
        rt_kprintf("mkdir ok!\n");
    }
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(mkdir_sample, mkdir sample);

```

在 FinSH 控制台运行该示例，运行结果如下：

```
msh />mkdir_sample
mkdir ok!
msh />ls
Directory /:
dir_test <DIR> # <DIR> 表示该目录的类型是文件夹
```

本例程演示了在根目录下创建名为 dir_test 的文件夹。

23.6.6 读取目录示例

本小节的示例代码展示如何读取目录，程序会创建一个操作文件的函数 `readdir_sample()` 并导出到 `msh` 命令列表，该函数会调用 `readdir()` 函数获取 `dir_test` 文件夹的内容信息并打印出来。示例代码如下所示：

```
#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

static void readdir_sample(void)
{
    DIR *dirp;
    struct dirent *d;

    /* 打开 / dir_test 目录 */
    dirp = opendir("/dir_test");
    if (dirp == RT_NULL)
    {
        rt_kprintf("open directory error!\n");
    }
    else
    {
        /* 读取目录 */
        while ((d = readdir(dirp)) != RT_NULL)
        {
            rt_kprintf("found %s\n", d->d_name);
        }

        /* 关闭目录 */
        closedir(dirp);
    }
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(readdir_sample, readdir sample);
```

在 FinSH 控制台运行该示例，运行结果如下：

```
msh />ls
Directory /:
```

```

dir_test          <DIR>
msh />cd dir_test
msh /dir_test>echo "hello" hello.txt      # 创建一个 hello.txt 文件
msh /dir_test>cd ..                         # 切换到上级文件夹
msh />readdir_sample
found hello.txt

```

本示例中，首先进入到 `dir_test` 文件夹下创建 `hello.txt` 文件，然后退出 `dir_test` 文件夹。此时运行示例程序将 `dir_test` 文件夹中的内容打印出来。

23.6.7 设置读取目录位置示例

本小节的示例代码展示如何设置下次读取目录的位置，程序会创建一个操作文件的函数 `telldir_sample()` 并导出到 `msh` 命令列表。该函数会首先打开根目录，然后读取根目录下所有目录信息，并将这些目录信息打印出来。同时使用 `telldir()` 函数记录第三个目录项的位置信息。在第二次读取根目录下的目录信息前，使用 `seekdir()` 函数设置读取位置为之前记录的第三个目录项的地址，此时再次读取根目录下的信息，并将目录信息打印出来。示例代码如下所示：

```

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

/* 假设文件操作是在一个线程中完成 */
static void telldir_sample(void)
{
    DIR *dirp;
    int save3 = 0;
    int cur;
    int i = 0;
    struct dirent *dp;

    /* 打开根目录 */
    rt_kprintf("the directory is:\n");
    dirp = opendir("/");

    for (dp = readdir(dirp); dp != RT_NULL; dp = readdir(dirp))
    {
        /* 保存第三个目录项的目录指针 */
        i++;
        if (i == 3)
            save3 = telldir(dirp);

        rt_kprintf("%s\n", dp->d_name);
    }

    /* 回到刚才保存的第三个目录项的目录指针 */
    seekdir(dirp, save3);

    /* 检查当前目录指针是否等于保存过的第三个目录项的指针。 */
}

```

```
cur = telldir(dirp);
if (cur != save3)
{
    rt_kprintf("seekdir (d, %ld); telldir (d) == %ld\n", save3, cur);
}

/* 从第三个目录项开始打印 */
rt_kprintf("the result of tell_seek_dir is:\n");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
{
    rt_kprintf("%s\n", dp->d_name);
}

/* 关闭目录 */
closedir(dirp);
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(telldir_sample, telldir sample);
```

本次演示示例中，需要手动在根目录下用 `mkdir` 命令依次创建从 `hello_1` 到 `hello_5` 这五个文件夹，确保根目录下有运行示例所需的文件夹目录。

在 FinSH 控制台运行该示例，运行结果如下：

```
msh />ls
Directory /:
hello_1          <DIR>
hello_2          <DIR>
hello_3          <DIR>
hello_4          <DIR>
hello_5          <DIR>
msh />telldir_sample
the directory is:
hello_1
hello_2
hello_3
hello_4
hello_5
the result of tell_seek_dir is:
hello_3
hello_4
hello_5
```

运行示例程序后，可以看到第一次读取根目录信息时是从第一个文件夹开始读取，打印出了根目录下所有的目录信息。第二次打印目录信息时，由于使用了 `seekdir()` 函数设置读取的起始位置为第三个文件夹的位置，因此第二次读取根目录时，是从第三个文件夹开始读取直到最后一个文件夹，只打印出了从 `hello_3` 到 `hello_5` 的目录信息。

23.7 常见问题

23.7.1 Q: 发现文件名或者文件夹名称显示不正常怎么办？

A: 检查是否开启了长文件名支持，DFS 功能配置小节。

23.7.2 Q: 文件系统初始化失败怎么办？

A: 检查文件系统配置项目中的允许挂载的文件系统类型和数量是否充足。

23.7.3 Q: 创建文件系统 `mkfs` 命令失败怎么办？

A: 检查存储设备是否存在，如果存在检查设备驱动是否可以通过功能测试，如果不能通过，则检查驱动错误。检查 `libc` 功能是否开启。

23.7.4 Q: 文件系统挂载失败怎么办？

A:

- 检查指定的挂载路径是否存在。文件系统可以直接挂载到根目录（“/”），但是如果想要挂载到其他路径上，如（“/sdcard”）。需要确保（“/sdcard”）路径是存在的，否则需要先在根目录创建 `sdcard` 文件夹才能挂载成功。
- 检查是否在存储设备上创建了文件系统，如果存储设备上没有文件系统，需要使用 `mkfs` 命令在存储器上创建文件系统。

23.7.5 Q: SFUD 探测不到 Flash 所使用的具体型号怎么办？

A:

- 检查硬件引脚设置是否错误。
- SPI 设备是否已经注册。
- SPI 设备是否已经挂载到总线。
- 检查在 RT-Thread Components □ Device Drivers -> Using SPI Bus/Device device drivers -> Using Serial Flash Universal Driver 菜单下的 Using auto probe flash JEDEC SDFP parameter 和 Using defined supported flash chip information table 配置项是否选中，如果没有选中那么需要开启这两个选项。
- 如果开启了上面的选项仍然无法识别存储设备，那么可以在 SFUD 项目中提出 issues。

23.7.6 Q: 存储设备的 **benchmark** 测试耗时过长是怎么回事？

A:

- 可对比 `system tick` 为 1000 时的 **benchmark** 测试数据 和本次测试所需的时长，如果耗时差距过大，则可以认为测试工作运行不正常。
- 检查系统 `tick` 的设置，因为一些延时操作会根据 `tick` 时间来决定，所以需要根据系统情况来设置合适的 `system tick` 值。如果系统的 `system tick` 值不低于 1000，则需要使用逻辑分析仪检查波形确定通信速率正常。

23.7.7 Q: SPI Flash 实现 elmfat 文件系统，如何保留部分扇区不被文件系统使用？

A: 可以使用 RT-Thread 提供的 `partition` 工具软件包为整个存储设备创建多个块设备，为创建的多个块设备分配不同的功能即可。

23.7.8 Q: 测试文件系统过程中程序卡住了怎么办？

A: 尝试使用调试器或者打印一些必要的调试信息，确定程序卡住的位置再提出问题。

23.7.9 Q: 如何一步步检查文件系统出现的问题？

A:

- 可以采用从底层到上层的方法来逐步排查问题。
- 首先检查存储设备是否注册成功，功能是否正常。
- 检查存储设备中是否创建了文件系统。
- 检查指定文件系统类型是否注册到 DFS 框架，经常要检查允许的文件系统类型和数量是否足够。
- 检查 DFS 是否初始化成功，这一步的初始化操作是纯软件的，因此出错的可能性不高。需要注意的是如果开启了组件自动初始化，就无需再次手动初始化。

第 24 章

AT 组件

24.1 AT 命令简介

AT 命令（AT Commands）最早是由发明拨号调制解调器（MODEM）的贺氏公司（Hayes）为了控制 MODEM 而发明的控制协议。后来随着网络带宽的升级，速度很低的拨号 MODEM 基本退出一般使用市场，但是 AT 命令保留下来。当时主要的移动电话生产厂家共同为 GSM 研制了一整套 AT 命令，用于控制手机的 GSM 模块。AT 命令在此基础上演化并加入 GSM 07.05 标准以及后来的 GSM 07.07 标准，实现比较健全的标准化。

在随后的 GPRS 控制、3G 模块等方面，均采用的 AT 命令来控制，AT 命令逐渐在产品开发中成为实际的标准。如今，AT 命令也广泛的应用于嵌入式开发领域，AT 命令作为主芯片和通讯模块的协议接口，硬件接口一般为串口，这样主控设备可以通过简单的命令和硬件设计完成多种操作。

AT 命令集是一种应用于 AT 服务器（AT Server）与 AT 客户端（AT Client）间的设备连接与数据通信的方式。其基本结构如下图所示：



图 24.1: AT 命令框架

1. 一般 AT 命令由三个部分组成，分别是：前缀、主体和结束符。其中前缀由字符 AT 构成；主体由命令、参数和可能用到的数据组成；结束符一般为 `<CR><LF>` ("\\r\\n")。
2. AT 功能的实现需要 AT Server 和 AT Client 两个部分共同完成。
3. AT Server 主要用于接收 AT Client 发送的命令，判断接收的命令及参数格式，并下发对应的响应数据，或者主动下发数据。

4. AT Client 主要用于发送命令、等待 AT Server 响应，并对 AT Server 响应数据或主动发送的数据进行解析处理，获取相关信息。
5. AT Server 和 AT Client 之间支持多种数据通讯的方式（UART、SPI 等），目前最常用的是串口 UART 通讯方式。
6. AT Server 向 AT Client 发送的数据分成两种：响应数据和 URC 数据。
 - 响应数据：AT Client 发送命令之后收到的 AT Server 响应状态和信息。
 - URC 数据：AT Server 主动发送给 AT Client 的数据，一般出现在一些特殊的情况，比如 WIFI 连接断开、TCP 接收数据等，这些情况往往需要用户做出相应操作。

随着 AT 命令的逐渐普及，越来越多的嵌入式产品上使用了 AT 命令，AT 命令作为主芯片和通讯模块的协议接口，硬件接口一般为串口，这样主控设备可以通过简单的命令和硬件设计完成多种操作。

虽然 AT 命令已经形成了一定的标准化，但是不同的芯片支持的 AT 命令并没有完全统一，这直接提高了用户使用的复杂性。对于 AT 命令的发送和接收以及数据的解析没有统一的处理方式。并且在使用 AT 设备连接网络时，只能通过命令完成简单的设备连接和数据收发功能，很难做到对上层网络应用接口的适配，不利于产品设备的开发。

为了方便用户使用 AT 命令，简单的适配不同的 AT 模块，RT-Thread 提供了 AT 组件用于 AT 设备的连接和数据通讯。AT 组件的实现包括客户端的和服务器两部分。

24.2 AT 组件简介

AT 组件是基于 RT-Thread 系统的 AT Server 和 AT Client 的实现，组件完成 AT 命令的发送、命令格式及参数判断、命令的响应、响应数据的接收、响应数据的解析、URC 数据处理等整个 AT 命令数据交互流程。

通过 AT 组件，设备可以作为 AT Client 使用串口连接其他设备发送并接收解析数据，可以作为 AT Server 让其他设备甚至电脑端连接完成发送数据的响应，也可以在本地 shell 启动 CLI 模式使设备同时支持 AT Server 和 AT Client 功能，该模式多用于设备开发调试。

AT 组件资源占用：

- AT Client 功能：4.6K ROM 和 2.0K RAM；
- AT Server 功能：4.0K ROM 和 2.5K RAM；
- AT CLI 功能：1.5K ROM，几乎没有使用 RAM。

整体看来，AT 组件资源占用极小，因此非常适用于资源有限的嵌入式设备中。AT 组件代码主要位于 `rt-thread/components/net/at/` 目录中。主要的功能包括如下，

AT Server 主要功能特点：

- 基础命令：实现多种通用基础命令（ATE、ATZ 等）；
- 命令兼容：命令支持忽略大小写，提高命令兼容性；
- 命令检测：命令支持自定义参数表达式，并实现对接收的命令参数自检测功能；

- 命令注册：提供简单的用户自定义命令添加方式，类似于 `finsh/msh` 命令添加方式；
- 调试模式：提供 AT Server CLI 命令行交互模式，主要用于设备调试。

AT Client 主要功能特点：

- URC 数据处理：完备的 URC 数据的处理方式；
- 数据解析：支持自定义响应数据的解析方式，方便获取响应数据中相关信息；
- 调试模式：提供 AT Client CLI 命令行交互模式，主要用于设备调试。
- AT Socket：作为 AT Client 功能的延伸，使用 AT 命令收发作为基础，实现标准的 BSD Socket API，完成数据的收发功能，使用户通过 AT 命令完成设备连网和数据通讯。
- 多客户端支持：AT 组件目前支持多客户端同时运行。

24.3 AT Server

24.3.1 AT Server 配置

当我们使用 AT 组件中的 AT Server 功能时需要在 `rtconfig.h` 中定义如下配置：

宏定义	描述
<code>RT_USING_AT</code>	开启 AT 组件
<code>AT_USING_SERVER</code>	开启 AT Server 功能
<code>AT_SERVER_DEVICE</code>	定义设备上 AT Server 功能使用的串口通讯设备名称，确保未被使用且设备名称唯一，例如 <code>uart3</code> 设备
<code>AT_SERVER_RECV_BUFF_LEN</code>	AT Server 设备最大接收数据的长度
<code>AT_CMD_END_MARK_CRLF</code>	判断接收命令的行结束符
<code>AT_USING_CLI</code>	开启服务器命令行交互模式
<code>AT_DEBUG</code>	开启 AT 组件 DEBUG 模式，可以显示更多调试日志信息
<code>AT_PRINT_RAW_CMD</code>	开启实时显示 AT 命令通信数据模式，方便调试

对于不同的 AT 设备，发送命令的行结束符的格式有几种：`"\r\n"`、`"\r"`、`"\n"`，用户需要根据 AT Server 连接的设备类型选用对应的行结束符，进而判断发送命令行的结束，定义的方式如下：

宏定义	结束符
<code>AT_CMD_END_MARK_CRLF</code>	<code>"\r\n"</code>
<code>AT_CMD_END_MARK_CR</code>	<code>"\r"</code>
<code>AT_CMD_END_MARK_LF</code>	<code>"\n"</code>

上面配置选项可以直接在 `rtconfig.h` 文件中添加使用，也可以通过组件包管理工具 Env 配置选项加入，ENV 中具体路径如下：

```
RT-Thread Components --->
  Network --->
    AT commands --->
      [*] Enable AT commands
      [*] Enable debug log output
      [*] Enable AT commands server
      (uart3) Server device name
      (256) The maximum length of server data accepted
            The commands new line sign (\r\n) --->
      [ ] Enable AT commands client
      [*] Enable command-line interface for AT commands
      [ ] Enable print RAW format AT command communication data
```

添加配置完成之后可以使用命令行重新生成工程，或使用 `scons` 来进行编译生成。

24.3.2 AT Server 初始化

配置开启 AT Server 配置之后，需要在启动时对它进行初始化，开启 AT Server 功能，如果程序中已经使用了组件自动初始化，则不再需要额外进行单独的初始化，否则需要在初始化任务中调用如下函数：

```
int at_server_init(void);
```

AT Server 初始化函数，属于应用层函数，需要在使用 AT Server 功能或者使用 AT Server CLI 功能前调用。`at_server_init()` 函数完成对 AT 命令存放数据段初始化、AT Server 设备初始化以及 AT Server 使用的信号量等资源的初始化，并创建 `at_server` 线程用于 AT Server 中数据的接收的解析。

AT Server 初始化成功之后，设备就可以作为 AT 服务器与 AT 客户端的串口设备连接并进行数据通讯，或者使用串口转化工具连接 PC，使 PC 端串口调试助手作为 AT 客户端与其进行数据通讯。

24.3.3 自定义 AT 命令添加方式

目前，不同厂家的 AT 设备使用的 AT 命令集的格式没有完全的统一的标准，所以 AT 组件中的 AT Server 只支持了部分基础通用 AT 命令，例如：ATE、AT+RST 等，这些命令只能满足设备基本操作，用户想使用更多功能需要针对不同 AT 设备完成自定义 AT Server 命令，AT 组件提供类似于 `finsh/msh` 命令添加方式的 AT 命令添加方式，方便用户实现需要的命令。

AT Server 目前默认支持的基础命令如下：

- AT: AT 测试命令；
- ATZ: 设备恢复出厂设置；
- AT+RST: 设备重启；
- ATE: ATE1 开启回显，ATE0 关闭回显；
- AT&L: 列出全部命令列表；
- AT+UART: 设置串口设置信息。

AT 命令根据传入的参数格式不同可以实现不同的功能，对于每个 AT 命令最多包含四种功能，如下所述：

- 测试功能：AT+<x>=? 用于查询命令参数格式及取值范围；
- 查询功能：AT+<x>? 用于返回命令参数当前值；
- 设置功能：AT+<x>=... 用于用户自定义参数值；
- 执行功能：AT+<x> 用于执行相关操作。

每个命令的四种功能并不需要全部实现，用户自定义添加 AT Server 命令时，可根据自己需求实现一种或几种上述功能函数，未实现的功能可以使用 `NULL` 表示，再通过自定义命令添加函数添加到基础命令列表，添加方式类似于 `finsh/msh` 命令添加方式，添加函数如下：

```
AT_CMD_EXPORT(_name_, _args_expr_, _test_, _query_, _setup_, _exec_);
```

参数	描述
name	AT 命令名称
_args_expr_	AT 命令参数表达式；（无参数为 <code>NULL</code> , <x> 中为必选参数, [] 中为可选参数）
test	AT 测试功能函数名；（无实现为 <code>NULL</code> ）
query	AT 查询功能函数名；（同上）
setup	AT 设置功能函数名；（同上）
exec	AT 执行功能函数名；（同上）

如下为 AT 命令注册示例，`AT+TEST` 命令存在两个参数，第一个参数为必选参数，第二个参数为可选参数，命令实现查询功能和执行功能：

```
static at_result_t at_test_exec(void)
{
    at_server_printfln("AT test commands execute!");

    return 0;
}
static at_result_t at_test_query(void)
{
    at_server_printfln("AT+TEST=1,2");

    return 0;
}

AT_CMD_EXPORT("AT+TEST", =<value1>[,<value2>], NULL, at_test_query, NULL,
at_test_exec);
```

24.3.4 AT Server API 接口

24.3.4.1 发送数据至客户端（不换行）

```
void at_server_printf(const char *format, ...);
```

该函数用于 AT Server 通过串口设备发送固定格式的数据到对应的 AT Client 串口设备上，数据结尾不带换行符。用于自定义 AT Server 中 AT 命令的功能函数中。

参数	描述
format	自定义输入数据的表达式
...	输入数据列表，为可变参数

24.3.4.2 发送数据至客户端（换行）

```
void at_server_printfln(const char *format, ...);
```

该函数用于 AT Server 通过串口设备发送固定格式的数据到对应的 AT Client 串口设备上，数据结尾带换行符。用于自定义 AT Server 中 AT 命令的功能函数中。

参数	描述
format	自定义输入数据的表达式
...	输入数据列表，为可变参数

24.3.4.3 发送命令执行结果至客户端

```
void at_server_print_result(at_result_t result);
```

该函数用于 AT Server 通过串口设备发送命令执行结果到对应的 AT Client 串口设备上。AT 组件提供多种固定的命令执行结果类型，自定义命令时可以直接使用函数返回结果；

参数	描述
result	命令执行结果类型

AT 组件中命令执行结果类型以枚举类型给出，如下表所示：

命令执行结果类型	解释
AT_RESULT_OK	命令执行成功
AT_RESULT_FAILE	命令执行失败

命令执行结果类型	解释
AT_RESULT_NULL	命令无返回结果
AT_RESULT_CMD_ERR	输入命令错误
AT_RESULT_CHECK_FAILE	参数表达式匹配错误
AT_RESULT_PARSE_FAILE	参数解析错误

可参考以下代码了解如何使用 at_server_print_result 函数:

```
static at_result_t at_test_setup(const char *args)
{
    if(!args)
    {
        /* 如果传入的命令之后的参数错误，返回表达式匹配错误结果 */
        at_server_print_result(AT_RESULT_CHECK_FAILE);
    }

    /* 正常情况下返回执行成功结果 */
    at_server_print_result(AT_RESULT_OK);
    return 0;
}

static at_result_t at_test_exec(void)
{
    // execute some functions of the AT command.

    /* 该命令不需要返回结果 */
    at_server_print_result(AT_RESULT_NULL);
    return 0;
}

AT_CMD_EXPORT("AT+TEST", <value1>, <value2>, NULL, NULL, at_test_setup, at_test_exec
);
```

24.3.4.4 解析输入命令参数

```
int at_req_parse_args(const char *req_args, const char *req_expr, ...);
```

一个 AT 命令的四种功能函数中，只有设置函数有入参，该入参为去除 AT 命令剩余部分，例如一个命令输入为 "AT+TEST=1,2,3,4"，则设置函数的入参为参数字符串 "=1,2,3,4" 部分。

该命令解析函数主要用于 AT 命令的设置函数中，用于解析传入字符串参数，得到对应的多个输入参数，用于执行后面操作，这里的解析语法使用的标准 `sscanf` 解析语法，后面 AT Client 参数解析函数中会详细介绍。

参数	描述
req_args	请求命令的传入参数字符串

参数	描述
req_expr	自定义参数解析表达式，用于解析上述传入参数数据
...	输出的解析参数列表，为可变参数
返回	-
>0	成功，返回匹配参数表达式的可变参数个数
=0	失败，无匹配参数表达式的参数
-1	失败，参数解析错误

可参考以下代码了解如何使用 at_server_print_result 函数：

```
static at_result_t at_test_setup(const char *args)
{
    int value1,value2;

    /* args 的输入标准格式应为 "=1,2"， "%d,%d" 为自定义参数解析表达式， 解析得到结果
       存入 value1 和 value2 变量 */
    if (at_req_parse_args(args, "%d,%d", &value1, &value2) > 0)
    {
        /* 数据解析成功，回显数据到 AT Server 串口设备 */
        at_server_printf("value1 : %d, value2 : %d", value1, value2);

        /* 数据解析成功，解析参数的个数大于零，返回执行成功 */
        at_server_print_result(AT_RESULT_OK);
    }
    else
    {
        /* 数据解析失败，解析参数的个数不大于零，返回解析失败结果类型 */
        at_server_print_result(AT_RESULT_PARSE_FAILE);
    }
    return 0;
}
/* 添加 "AT+TEST" 命令到 AT 命令列表，命令参数格式为两个必选参数 <value1> 和 <value2> */
AT_CMD_EXPORT("AT+TEST", <value1>,<value2>, NULL, NULL, at_test_setup, NULL);
```

移植相关接口

AT Server 默认已支持多种基础命令（ATE、ATZ 等），其中部分命令的函数实现与硬件或平台相关，需要用户自定义实现。AT 组件源码 `src/at_server.c` 文件中给出了移植文件的弱函数定义，用户可在项目中新建移植文件实现如下函数完成移植接口，也可以直接在文件中修改弱函数完成移植接口。

1. 设备重启函数:`void at_port_reset(void);`。该函数完成设备软重启功能，用于 AT Server 中基础命令 AT+RST 的实现。
2. 设备恢复出厂设置函数：`void at_port_factory_reset(void);`。该函数完成设备恢复出厂设置功能，用于 AT Server 中基础命令 ATZ 的实现。

3. 链接脚本中添加命令表（gcc 添加，keil、iar 跳过）

工程中若使用 `gcc` 工具链，需在链接脚本中添加 AT 服务器命令表对应的 section，参考如下链接脚本：

```
/* Constant data goes into FLASH */
.rodata :
{
    ...

    /* section information for RT-thread AT package */
    . = ALIGN(4);
    __rtatcmdtab_start = .;
    KEEP(*(RtAtCmdTab))
    __rtatcmdtab_end = .;
    . = ALIGN(4);
} > CODE
```

24.4 AT Client

24.4.1 AT Client 配置

当我们使用 AT 组件中的 AT Client 功能时需要在 `rtconfig.h` 中定义如下配置：

```
#define RT_USING_AT
#define AT_USING_CLIENT
#define AT_CLIENT_NUM_MAX 1
#define AT_USING_SOCKET
#define AT_USING_CLI
#define AT_PRINT_RAW_CMD
```

- `RT_USING_AT`: 用于开启或关闭 AT 组件；
- `AT_USING_CLIENT`: 用于开启 AT Client 功能；
- `AT_CLIENT_NUM_MAX`: 最大同时支持的 AT 客户端数量。
- `AT_USING_SOCKET`: 用于 AT 客户端支持标准 BSD Socket API，开启 AT Socket 功能。
- `AT_USING_CLI`: 用于开启或关闭客户端命令行交互模式。
- `AT_PRINT_RAW_CMD`: 用于开启 AT 命令通信数据的实时显示模式，方便调试

上面配置选项可以直接在 `rtconfig.h` 文件中添加使用，也可以通过组件包管理工具 Env 配置选项加入，ENV 中具体路径如下：

```
RT-Thread Components --->
    Network --->
        AT commands --->
```

```
[*] Enable AT commands
[ ]  Enable debug log output
[ ]  Enable AT commands server
[*]  Enable AT commands client
(1)  The maximum number of supported clients
[*]  Enable BSD Socket API support by AT commands
[*]  Enable command-line interface for AT commands
[ ]  Enable print RAW format AT command communication data
```

添加配置完成之后可以使用命令行重新生成工程，或使用 `scons` 来进行编译生成。

24.4.2 AT Client 初始化

配置开启 AT Client 配置之后，需要在启动时对它进行初始化，开启 AT client 功能，如果程序中已经使用了组件自动初始化，则不再需要额外进行单独的初始化，否则需要在初始化任务中调用如下函数：

```
int at_client_init(const char *dev_name, rt_size_t recv_bufsz);
```

AT Client 初始化函数，属于应用层函数，需要在使用 AT Client 功能或者使用 AT Client CLI 功能前调用。`at_client_init()` 函数完成对 AT Client 设备初始化、AT Client 移植函数的初始化、AT Client 使用的信号量、互斥锁等资源初始化，并创建 `at_client` 线程用于 AT Client 中数据的接收的解析以及对 URC 数据的处理。

24.4.3 AT Client 数据收发方式

AT Client 主要功能是发送 AT 命令、接收数据并解析数据。下面是对 AT Client 数据接收和发送相关流程与函数介绍。相关结构体定义：

```
struct at_response
{
    /* response buffer */
    char *buf;
    /* the maximum response buffer size */
    rt_size_t buf_size;
    /* the number of setting response lines
     * == 0: the response data will auto return when received 'OK' or 'ERROR'
     * != 0: the response data will return when received setting lines number data
     */
    rt_size_t line_num;
    /* the count of received response lines */
    rt_size_t line_counts;
    /* the maximum response time */
    rt_int32_t timeout;
};

typedef struct at_response *at_response_t;
```

AT 组件中，该结构体用于定义一个 AT 命令响应数据的控制块，用于存放或者限制 AT 命令响应数据的数据格式。其中 `buf` 用于存放接收到的响应数据，注意的是 `buf` 中存放的数据并不是原始响应数据，而

是原始响应数据去除结束符（"\r\n"）的数据，**buf** 中每行数据以 '\0' 分割，方便按行获取数据。**buf_size** 为用户自定义本次响应最大支持的接收数据的长度，由用户根据自己命令返回值长度定义。**line_num** 为用户自定义的本次响应数据需要接收的行数，如果没有响应行数限定需求，可以置为 0。**line_counts** 用于记录本次响应数据总行数。**timeout** 为用户自定义的本次响应数据最大响应时间。该结构体中 **buf_size**、**line_num**、**timeout** 三个参数为限制条件，在结构体创建时设置，其他参数为存放数据参数，用于后面数据解析。

相关 API 接口介绍：

24.4.3.1 创建响应结构体

```
at_response_t at_create_resp(rt_size_t buf_size, rt_size_t line_num, rt_int32_t timeout);
```

参数	描述
buf_size	本次响应最大支持的接收数据的长度
line_num	本次响应需要返回数据的行数，行数是以标准结束符（如 "\r\n"）划分。若为 0，则接收到“OK”或“ERROR”数据后结束本次响应接收；若大于 0，接收完当前设置行号的数据后返回成功
timeout	本次响应数据最大响应时间，数据接收超时返回错误
返回	-
!= NULL	成功，返回指向响应结构体的指针
= NULL	失败，内存不足

该函数用于创建自定义的响应数据接收结构，用于后面接收并解析发送命令响应数据。

24.4.3.2 删除响应结构体

```
void at_delete_resp(at_response_t resp);
```

参数	描述
resp	准备删除的响应结构体指针

该函数用于删除创建的响应结构体对象，一般与 **at_create_resp** 创建函数成对出现。

24.4.3.3 设置响应结构体参数

```
at_response_t at_resp_set_info(at_response_t resp, rt_size_t buf_size, rt_size_t line_num, rt_int32_t timeout);
```

参数	描述
resp	已经创建的响应结构体指针
buf_size	本次响应最大支持的接收数据的长度
line_num	本次响应需要返回数据的行数，行数是以标准结束符划分若为 0，则接收到“OK”或“ERROR”数据后结束本次响应接收若大于 0，接收完当前设置行号的数据后返回成功
timeout	本次响应数据最大响应时间，数据接收超时返回错误
返回	-
!= NULL	成功，返回指向响应结构体的指针
= NULL	失败，内存不足

该函数用于设置已经创建的响应结构体信息，主要设置对响应数据的限制信息，一般用于创建结构体之后，发送 AT 命令之前。该函数主要用于设备初始化时命令的发送，可以减少响应结构体创建次数，降低代码资源占用。

24.4.3.4 发送命令并接收响应

```
rt_err_t at_exec_cmd(at_response_t resp, const char *cmd_expr, ...);
```

参数	描述
resp	创建的响应结构体指针
cmd_expr	自定义输入命令的表达式
...	输入命令数据列表，为可变参数
返回	-
>=0	成功
-1	失败
-2	失败，接收响应超时

该函数用于 AT Client 发送命令到 AT Server，并等待接收响应，其中 `resp` 是已经创建好的响应结构体的指针，AT 命令的使用匹配表达式的可变参输入，输入命令的结尾不需要添加命令结束符。

可参考以下代码了解如何使用以上几个 AT 命令收发相关函数使用方式：

```
/*
 * 程序清单：AT Client 发送命令并接收响应例程
 */

#include <rtthread.h>
#include <at.h> /* AT 组件头文件 */
```

```
int at_client_send(int argc, char**argv)
{
    at_response_t resp = RT_NULL;

    if (argc != 2)
    {
        LOG_E("at_cli_send [command] - AT client send commands to AT server.");
        return -RT_ERROR;
    }

    /* 创建响应结构体，设置最大支持响应数据长度为 512 字节，响应数据行数无限制，超时时间为 5 秒 */
    resp = at_create_resp(512, 0, rt_tick_from_millisecond(5000));
    if (!resp)
    {
        LOG_E("No memory for response structure!");
        return -RT_ENOMEM;
    }

    /* 发送 AT 命令并接收 AT Server 响应数据，数据及信息存放在 resp 结构体中 */
    if (at_exec_cmd(resp, argv[1]) != RT_EOK)
    {
        LOG_E("AT client send commands failed, response error or timeout !");
        return -ET_ERROR;
    }

    /* 命令发送成功 */
    LOG_D("AT Client send commands to AT Server success!");

    /* 删除响应结构体 */
    at_delete_resp(resp);

    return RT_EOK;
}

#ifndef FINSH_USING_MSH
#include <finsh.h>
/* 输出 at_Client_send 函数到 msh 中 */
MSH_CMD_EXPORT(at_Client_send, AT Client send commands to AT Server and get response
               data);
#endif
```

发送和接收数据的实现原理比较简单，主要是对 AT Client 绑定的串口设备的读写操作，并设置相关行数和超时来限制响应数据，值得注意的是，正常情况下需要先创建 resp 响应结构体传入 at_exec_cmd 函数用于数据的接收，当 at_exec_cmd 函数传入 resp 为 NULL 时说明本次发送数据不考虑处理响应数据直接返回结果。

24.4.4 AT Client 数据解析方式

数据正常获取之后，需要对响应的数据进行解析处理，这也是 AT Client 重要的功能之一。AT Client 中数据的解析提供自定义解析表达式的解析形式，其解析语法使用标准的 `sscanf` 解析语法。开发者可以通过自定义数据解析表达式回去响应数据中有用信息，前提是开发者需要提前查看相关手册了解 AT Client 连接的 AT Server 设备响应数据的基本格式。下面通过几个函数和例程简单 AT Client 数据解析方式。

24.4.4.1 获取指定行号的响应数据

```
const char *at_resp_get_line(at_response_t resp, rt_size_t resp_line);
```

参数	描述
resp	响应结构体指针
resp_line	需要获取数据的行号
返回	-
<code>!= NULL</code>	成功，返回对应行号数据的指针
<code>= NULL</code>	失败，输入行号错误

该函数用于在 AT Server 响应数据中获取指定行号的一行数据。行号是以标准数据结束符来判断的，上述发送和接收函数 `at_exec_cmd` 已经对响应数据的数据和行号进行记录处理存放于 `resp` 响应结构体中，这里可以直接获取对应行号的数据信息。

24.4.4.2 获取指定关键字的响应数据

```
const char *at_resp_get_line_by_kw(at_response_t resp, const char *keyword);
```

参数	描述
resp	响应结构体指针
keyword	关键字信息
返回	-
<code>!= NULL</code>	成功，返回对应行号数据的指针
<code>= NULL</code>	失败，未找到关键字信息

该函数用于在 AT Server 响应数据中通过关键字获取对应的一行数据。

24.4.4.3 解析指定行号的响应数据

```
int at_resp_parse_line_args(at_response_t resp, rt_size_t resp_line, const char *
    resp_expr, ...);
```

参数	描述
resp	响应结构体指针
resp_line	需要解析数据的行号, 行号从 1 开始计数
resp_expr	自定义的参数解析表达式
...	解析参数列表, 为可变参数
返回	-
>0	成功, 返回解析成功的参数个数
=0	失败, 无匹配数解析表达式的参数
-1	失败, 参数解析错误

该函数用于在 AT Server 响应数据中获取指定行号的一行数据, 并解析该行数据中的参数。

24.4.4.4 解析指定关键字行的响应数据

```
int at_resp_parse_line_args_by_kw(at_response_t resp, const char *keyword, const
    char *resp_expr, ...);
```

参数	描述
resp	响应结构体指针
keyword	关键字信息
resp_expr	自定义的参数解析表达式
...	解析参数列表, 为可变参数
返回	-
>0	成功, 返回解析成功的参数个数
=0	失败, 无匹配数解析表达式的参数
-1	失败, 参数解析错误

该函数用于在 AT Server 响应数据中获取包含关键字的一行数据, 并解析该行数据中的参数。

数据解析语法使用标准 `sscanf` 解析语法, 语法的内容比较多, 开发者可以自行搜索其解析语法, 这里使用两个例程介绍简单使用方法。

24.4.4.5 串口配置信息解析示例

客户端发送的数据:

```
AT+UART?
```

客户端获取的响应数据:

```
UART=115200,8,1,0,0\r\n
OK\r\n
```

解析伪代码如下:

```
/* 创建服务器响应结构体，64 为用户自定义接收数据最大长度 */
resp = at_create_resp(64, 0, rt_tick_from_millisecond(5000));

/* 发送数据到服务器，并接收到响应数据存放在 resp 结构体中 */
at_exec_cmd(resp, "AT+UART?");

/* 解析获取串口配置信息，1 表示解析响应数据第一行，'%*[^=]' 表示忽略等号之前的数据 */
at_resp_parse_line_args(resp, 1, "%*[^=]=%d,%d,%d,%d", &baudrate, &databits, &
    stopbits, &parity, &control);
printf("baudrate=%d, databits=%d, stopbits=%d, parity=%d, control=%d\n",
    baudrate, databits, stopbits, parity, control);

/* 删除服务器响应结构体 */
at_delete_resp(resp);
```

24.4.4.6 IP 和 MAC 地址解析示例

客户端发送的数据:

```
AT+IPMAC?
```

服务器获取的响应数据:

```
IP=192.168.1.10\r\n
MAC=12:34:56:78:9a:bc\r\n
OK\r\n
```

解析伪代码如下:

```
/* 创建服务器响应结构体，128 为用户自定义接收数据最大长度 */
resp = at_create_resp(128, 0, rt_tick_from_millisecond(5000));

at_exec_cmd(resp, "AT+IPMAC?");

/* 自定义解析表达式，解析当前行号数据中的信息 */
at_resp_parse_line_args(resp, 1, "IP=%s", ip);
at_resp_parse_line_args(resp, 2, "MAC=%s", mac);
```

```
printf("IP=%s, MAC=%s\n", ip, mac);

at_delete_resp(resp);
```

解析数据的关键在于解析表达式的正确定义，因为对于 AT 设备的响应数据，不同设备厂家不同命令的响应数据格式不唯一，所以只能提供自定义解析表达式的形式获取需要信息，`at_resp_parse_line_args` 解析参数函数的设计基于 `sscanf` 数据解析方式，开发者使用之前需要先了解基本的解析语法，再结合响应数据设计合适的解析语法。如果开发者不需要解析具体参数，可以直接使用 `at_resp_get_line` 函数获取一行的具体数据。

24.4.5 AT Client URC 数据处理

URC 数据的处理是 AT Client 另一个重要功能，URC 数据为服务器主动下发的数据，不能通过上述数据发送接收函数接收，并且对于不同设备 URC 数据格式和功能不一样，所以 URC 数据处理的方式也是需要用户自定义实现的。AT 组件中对 URC 数据的处理提供列表管理方式，用户可自定义添加 URC 数据及其执行函数到管理列表中，所以 URC 数据的处理也是 AT Client 的主要移植工作。

相关结构体：

```
struct at_urc
{
    const char *cmd_prefix;           // URC 数据前缀
    const char *cmd_suffix;           // URC 数据后缀
    void (*func)(const char *data, rt_size_t size); // URC 数据执行函数
};

typedef struct at_urc *at_urc_t;
```

每种 URC 数据都有一个结构体控制块，用于定义判断 URC 数据的前缀和后缀，以及 URC 数据的执行函数。一段数据只有完全匹配 URC 的前缀和后缀才能定义为 URC 数据，获取到匹配的 URC 数据后会立刻执行 URC 数据执行函数。所以开发者添加一个 URC 数据需要自定义匹配的前缀、后缀和执行函数。

24.4.5.1 URC 数据列表初始化

```
void at_set_urc_table(const struct at_urc *table, rt_size_t size);
```

参数	描述
table	URC 数据结构体数组指针
size	URC 数据的个数

该函数用于初始化开发者自定义的 URC 数据列表，主要在 AT Client 移植函数中使用。

下面给出 AT Client 移植具体示例，该示例主要展示 `at_client_port_init()` 移植函数中 URC 数据的具体处理方式，开发者可直接应用到自己的移植文件中，或者自定义修改实现功能，完成 AT Client 的移植。

```

static void urc_conn_func(const char *data, rt_size_t size)
{
    /* WIFI 连接成功信息 */
    LOG_D("AT Server device WIFI connect success!");
}

static void urc_recv_func(const char *data, rt_size_t size)
{
    /* 接收到服务器发送数据 */
    LOG_D("AT Client receive AT Server data!");
}

static void urc_func(const char *data, rt_size_t size)
{
    /* 设备启动信息 */
    LOG_D("AT Server device startup!");
}

static struct at_urc urc_table[] = {
    {"WIFI CONNECTED",    "\r\n",      urc_conn_func},
    {"+RECV",              ":",        urc_recv_func},
    {"RDY",                 "\r\n",      urc_func},
};

int at_client_port_init(void)
{
    /* 添加多种 URC 数据至 URC 列表中，当接收到同时匹配 URC 前缀和后缀的数据，执行
       URC 函数 */
    at_set_urc_table(urc_table, sizeof(urc_table) / sizeof(urc_table[0]));
    return RT_EOK;
}

```

24.4.6 AT Client 其他 API 接口介绍

24.4.6.1 发送指定长度数据

```
rt_size_t at_client_send(const char *buf, rt_size_t size);
```

参数	描述
buf	发送数据的指针
size	发送数据的长度
返回	-
>0	成功，返回发送成功的数据长度
<=0	失败

该函数用于通过 AT Client 设备发送指定长度数据到 AT Server 设备，多用于 AT Socket 功能。

24.4.6.2 接收指定长度数据

```
rt_size_t at_client_recv(char *buf, rt_size_t size, rt_int32_t timeout);
```

参数	描述
buf	接收数据的指针
size	最大支持接收数据的长度
timeout	接收数据超时时间，单位为 tick
返回	-
>0	成功，返回接收成功的数据长度
<=0	失败，接收数据错误或超时

该函数用于通过 AT Client 设备接收指定长度的数据，多用于 AT Socket 功能。该函数只能在 URC 回调处理函数中使用。

24.4.6.3 设置接收数据的行结束符

```
void at_set_end_sign(char ch);
```

参数	描述
ch	行结束符
返回	描述
无	无

该函数用于设置行结束符，用于判断客户端接收一行数据的结束，多用于 AT Socket 功能。

24.4.6.4 等待模块初始化完成

```
int at_client_wait_connect(rt_uint32_t timeout);
```

参数	描述
timeout	等待超时时间
返回	描述
0	成功

参数	描述
<0	失败，超时时间内无数据返回

该函数用于 AT 模块启动时循环发送 AT 命令，直到模块响应数据，说明模块启动成功。

24.4.7 AT Client 多客户端支持

一般情况下，设备作为 AT Client 只连接一个 AT 模块（AT 模块作为 AT Server）可直接使用上述数据收发和命令解析的函数。少数情况，设备作为 AT Client 需要连接多个 AT 模块，这种情况下需要设备的多客户端支持功能。

AT 组件提供对多客户端连接的支持，并且提供两套不同的函数接口：**单客户端模式函数**和**多客户端模式函数**。

- 单客户端模式函数：该类函数接口主要用于设备只连接一个 AT 模块情况，或者在设备连接多个 AT 模块时，用于第一个初始化的 AT 客户端中。
- 多客户端模式函数：该类函数接口主要用设备连接多个 AT 模块情况。

两种不同模式函数和在不同应用场景下的优缺点如下图：

应用场景 函数模式	单客户端模式函数	多客户端模式函数
单个模块	可直接使用，简单方便	优：支持单个模块场景，提高兼容性 缺：需要先获取客户端对象，使用流程比单客户端模式复杂
多个模块	优：简化函数操作流程 缺：多模块情况该函数模式只能用于第一个初始化的模块	需要先获取客户端对象，然后直接使用

图 24.2: at client 模式对比图

单客户端模式函数定义与单连接模式函数相比，主要是对传入的客户端对象的定义不同，单客户端模式函数默认使用第一个初始化的 AT 客户端对象，多客户端模式函数可以传入用户自定义获取的客户端对象，获取客户端对象的函数如下：

```
at_client_t at_client_get(const char *dev_name);
```

该函数通过传入的设备名称获取该设备创建的 AT 客户端对象，用于多客户端连接时区分不同的客户端。

单客户端模式和多客户端模式函数接口定义区别如下几个函数：

单客户端模式函数	多客户端模式函数
at_exec_cmd(...)	at_obj_exec_cmd(client, ...)
at_set_end_sign(...)	at_obj_set_end_sign(client, ...)
at_set_urc_table(...)	at_obj_set_urc_table(client, ...)
at_client_wait_connect(...)	at_client_obj_wait_connect(client, ...)
at_client_send(...)	at_client_obj_send(client, ...)
at_client_recv(...)	at_client_obj_recv(client, ...)

两种模式客户端数据收发和解析的方式基本相同，在函数使用流程上有所不同，如下所示：

```
/* 单客户端模式函数使用方式 */

at_response_t resp = RT_NULL;

at_client_init("uart2", 512);

resp = at_create_resp(256, 0, 5000);

/* 使用单客户端模式函数发送命令 */
at_exec_cmd(resp, "AT+CIFSR");

at_delete_resp(resp);
```

```
/* 多客户端模式函数使用方式 */

at_response_t resp = RT_NULL;
at_client_t client = RT_NULL;

/* 初始化两个 AT 客户端 */
at_client_init("uart2", 512);
at_client_init("uart3", 512);

/* 通过名称获取对应的 AT 客户端对象 */
client = at_client_get("uart3");

resp = at_create_resp(256, 0, 5000);

/* 使用多客户端模式函数发送命令 */
at_obj_exec_cmd(client, resp, "AT+CIFSR");

at_delete_resp(resp);
```

其他函数使用的流程区别类似于上述 `at_obj_exec_cmd()` 函数，主要是先通过 `at_client_get()` 函数获取客户端对象，再通过传入的对象判断是哪个客户端，实现多客户端的支持。

24.5 常见问题

24.5.1 Q: 开启 AT 命令收发数据实时打印功能，shell 上日志显示错误怎么办？

A: 提高 shell 对应串口设备波特率为 921600，提高串口打印速度，防止数据量过大时打印显示错误。

24.5.2 Q: AT Socket 功能启动时，编译提示 “The AT socket device is not selected, please select it through the env menuconfig”？

A: 该错误因为开启 AT Socket 功能之后，默认开启 at device 软件包中为配置对应的设备型号，进入 at device 软件包，配置设备为 ESP8266 设备，配置 WIFI 信息，重新 scons 生成工程，编译下载。

24.5.3 Q: AT Socket 功能数据接收超时或者数据接收不全？

A: 该错误可能是 AT 使用的串口设备中接收数据缓冲区过小（RT_SERIAL_RB_BUFSZ 默认为 64 bytes），数据未及时接收完就被覆盖导致的，适当增加串口接收数据的缓冲区大小（如 256 bytes）。

第 25 章

SAL 套接字抽象层

25.1 SAL 简介

为了适配更多的网络协议栈类型，避免系统对单一网络协议栈的依赖，RT-Thread 系统提供了一套 SAL（套接字抽象层）组件，该组件完成对不同网络协议栈或网络实现接口的抽象并对上层提供一组标准的 BSD Socket API，这样开发者只需要关心和使用网络应用层提供的网络接口，而无需关心底层具体网络协议栈类型和实现，极大的提高了系统的兼容性，方便开发者完成协议栈的适配和网络相关的开发。SAL 组件主要功能特点：

- 抽象、统一多种网络协议栈接口；
- 提供 Socket 层面的 TLS 加密传输特性；
- 支持标准 BSD Socket API；
- 统一的 FD 管理，便于使用 read/write poll/select 来操作网络功能；

25.1.1 SAL 网络框架

RT-Thread 的网络框架结构如下所示：

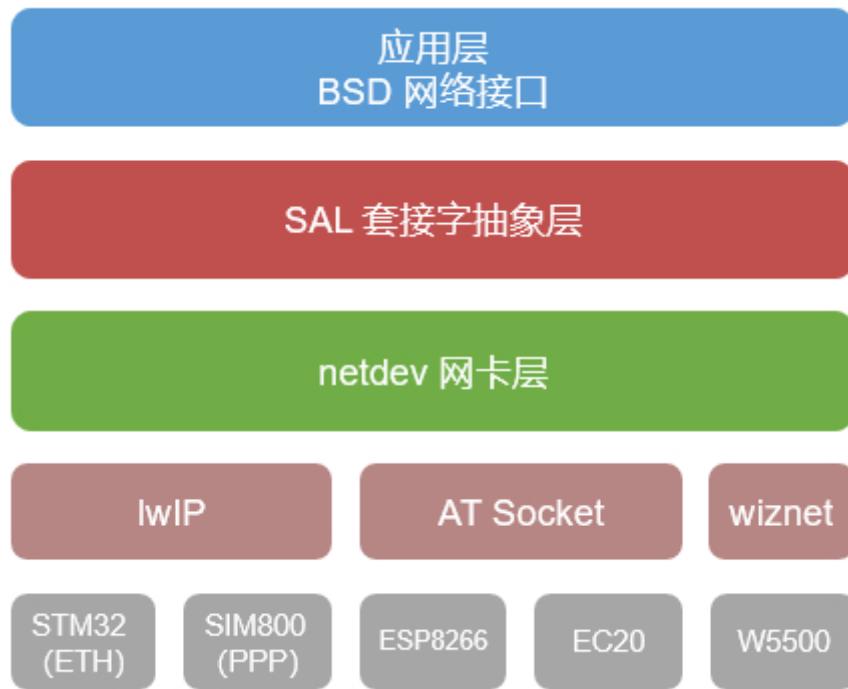


图 25.1: 网络框架图

最顶层是网络应用层，提供一套标准 BSD Socket API，如 `socket`、`connect` 等函数，用于系统中大部分网络开发应用。

往下第二部分为 SAL 套接字抽象层，通过它 RT-Thread 系统能够适配下层不同的网络协议栈，并提供给上层统一的网络编程接口，方便不同协议栈的接入。套接字抽象层为上层应用层提供接口有：`accept`、`connect`、`send`、`recv` 等。

第三部分为 netdev 网卡层，主要作用是解决多网卡情况设备网络连接和网络管理相关问题，通过 netdev 网卡层用户可以统一管理各个网卡信息和网络连接状态，并且可以使用统一的网卡调试命令接口。

第四部分为协议栈层，该层包括几种常用的 TCP/IP 协议栈，例如嵌入式开发中常用的轻型 TCP/IP 协议栈 lwIP 以及 RT-Thread 自主研发的 AT Socket 网络功能实现等。这些协议栈或网络功能实现直接和硬件接触，完成数据从网络层到传输层的转化。

RT-Thread 的网络应用层提供的接口主要以标准 BSD Socket API 为主，这样能确保程序可以在 PC 上编写、调试，然后再移植到 RT-Thread 操作系统上。

25.1.2 工作原理

SAL 组件工作原理的介绍主要分为如下三部分：

- 多协议栈接入与接口函数统一抽象功能；
- SAL TLS 加密传输功能；

25.1.2.1 多协议栈接入与接口函数统一抽象功能

对于不同的协议栈或网络功能实现，网络接口的名称可能各不相同，以 `connect` 连接函数为例，lwIP 协议栈中接口名称为 `lwip_connect`，而 AT Socket 网络实现中接口名称为 `at_connect`。SAL 组件提供对不同协议栈或网络实现接口的抽象和统一，组件在 `socket` 创建时通过判断传入的协议簇（`domain`）类型来判断使用的协议栈或网络功能，完成 RT-Thread 系统中多协议的接入与使用。

目前 SAL 组件支持的协议栈或网络实现类型有：**lwIP 协议栈**、**AT Socket 协议栈**、**WIZnet 硬件 TCP/IP 协议栈**。

```
int socket(int domain, int type, int protocol);
```

上述为标准 BSD Socket API 中 `socket` 创建函数的定义，`domain` 表示协议域又称为协议簇（`family`），用于判断使用哪种协议栈或网络实现，AT Socket 协议栈使用的簇类型为 `AF_AT`，lwIP 协议栈使用协议簇类型有 `AF_INET` 等，WIZnet 协议栈使用的协议簇类型为 `AF_WIZ`。

对于不同的软件包，`socket` 传入的协议簇类型可能是固定的，不会随着 SAL 组件接入方式的不同而改变。为了动态适配不同协议栈或网络实现的接入，SAL 组件中对于每个协议栈或者网络实现提供两种协议簇类型匹配方式：主协议簇类型和次协议簇类型。`socket` 创建时先判断传入协议簇类型是否存在已经支持的主协议类型，如果是则使用对应协议栈或网络实现，如果不是判断次协议簇类型是否支持。目前系统支持协议簇类型如下：

```
lwIP 协议栈: family = AF_INET、sec_family = AF_INET
AT Socket 协议栈: family = AF_AT、sec_family = AF_INET
WIZnet 硬件 TCP/IP 协议栈: family = AF_WIZ、sec_family = AF_INET
```

SAL 组件主要作用是统一抽象底层 BSD Socket API 接口，下面以 `connect` 函数调用流程为例说明 SAL 组件函数调用方式：

- `connect`: SAL 组件对外提供的抽象的 BSD Socket API，用于统一 `fd` 管理；
- `sal_connect`: SAL 组件中 `connect` 实现函数，用于调用底层协议栈注册的 `operation` 函数。
- `lwip_connect`: 底层协议栈提供的层 `connect` 连接函数，在网卡初始化完成时注册到 SAL 组件中，最终调用的操作函数。

```
/* SAL 组件为应用层提供的标准 BSD Socket API */
int connect(int s, const struct sockaddr *name, socklen_t namelen)
{
    /* 获取 SAL 套接字描述符 */
    int socket = dfs_net_getsocket(s);

    /* 通过 SAL 套接字描述符执行 sal_connect 函数 */
    return sal_connect(socket, name, namelen);
}

/* SAL 组件抽象函数接口实现 */
int sal_connect(int socket, const struct sockaddr *name, socklen_t namelen)
{
    struct sal_socket *sock;
    struct sal_proto_family *pf;
```

```

int ret;

/* 检查 SAL socket 结构体是否正常 */
SAL_SOCKET_OBJ_GET(sock, socket);

/* 检查当前 socket 网络连接状态是否正常 */
SAL_NETDEV_IS_COMMONICABLE(sock->netdev);
/* 检查当前 socket 对应的底层 operation 函数是否正常 */
SAL_NETDEV_SOCKETOPS_VALID(sock->netdev, pf, connect);

/* 执行底层注册的 connect operation 函数 */
ret = pf->skt_ops->connect((int) sock->user_data, name, namelen);
#ifndef SAL_USING_TLS
    if (ret >= 0 && SAL_SOCKOPS_PROTO_TLS_VALID(sock, connect))
    {
        if (proto_tls->ops->connect(sock->user_data_tls) < 0)
        {
            return -1;
        }
        return ret;
    }
#endif
    return ret;
}

/* lwIP 协议栈函数底层 connect 函数实现 */
int lwip_connect(int socket, const struct sockaddr *name, socklen_t namelen)
{
    ...
}

```

25.1.2.2 SAL TLS 加密传输功能

1. SAL TLS 功能介绍

在 TCP、UDP 等协议数据传输时，由于数据包是明文的，所以很可能被其他人拦截并解析出信息，这给信息的安全传输带来很大的影响。为了解决此类问题，一般需要用户在应用层和传输层之间添加 SSL/TLS 协议。

TLS (Transport Layer Security, 传输层安全协议) 是建立在传输层 TCP 协议之上的协议，其前身是 SSL (Secure Socket Layer, 安全套接字层)，主要作用是将应用层的报文进行非对称加密后再由 TCP 协议进行传输，实现了数据的加密安全交互。

目前常用的 TLS 方式：**MbedTLS**、**OpenSSL**、**s2n** 等，但是对于不同的加密方式，需要使用其指定的加密接口和流程进行加密，对于部分应用层协议的移植较为复杂。因此 SAL TLS 功能产生，主要作用是提供 **Socket** 层面的 **TLS** 加密传输特性，抽象多种 **TLS** 处理方式，提供统一的接口用于完成 **TLS** 数据交互。

2. SAL TLS 功能使用方式

使用流程如下：

- 配置开启任意网络协议栈支持（如 lwIP 协议栈）；
- 配置开启 MbedTLS 软件包（目前只支持 MbedTLS 类型加密方式）；
- 配置开启 SAL_TLS 功能支持（如下配置选项章节所示）；

配置完成之后，只要在 socket 创建时传入的 protocol 类型使用 **PROTOCOL_TLS** 或 **PROTOCOL_DTLS**，即可使用标准 BSD Socket API 接口，完成 TLS 连接的建立和数据的收发。示例代码如下所示：

```
#include <stdio.h>
#include <string.h>

#include <rtthread.h>
#include <sys/socket.h>
#include <netdb.h>

/* RT-Thread 官网，支持 TLS 功能 */
#define SAL_TLS_HOST      "www.rt-thread.org"
#define SAL_TLS_PORT      443
#define SAL_TLS_BUFSZ     1024

static const char *send_data = "GET /download/rt-thread.txt HTTP/1.1\r\n"
    "Host: www.rt-thread.org\r\n"
    "User-Agent: rtthread/4.0.1 rtt\r\n\r\n";

void sal_tls_test(void)
{
    int ret, i;
    char *recv_data;
    struct hostent *host;
    int sock = -1, bytes_received;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host = gethostbyname(SAL_TLS_HOST);

    recv_data = rt_calloc(1, SAL_TLS_BUFSZ);
    if (recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
    }

    /* 创建一个socket，类型是SOCKET_STREAM，TCP 协议，TLS 类型 */
    if ((sock = socket(AF_INET, SOCK_STREAM, PROTOCOL_TLS)) < 0)
    {
        rt_kprintf("Socket error\n");
    }
}
```

```
        goto __exit;
    }

/* 初始化预连接的服务端地址 */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SAL_TLS_PORT);
server_addr.sin_addr = *((struct in_addr *)host->h_addr);
rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

if (connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) < 0)
{
    rt_kprintf("Connect fail!\n");
    goto __exit;
}

/* 发送数据到 socket 连接 */
ret = send(sock, send_data, strlen(send_data), 0);
if (ret <= 0)
{
    rt_kprintf("send error,close the socket.\n");
    goto __exit;
}

/* 接收并打印响应的数据，使用加密数据传输 */
bytes_received = recv(sock, recv_data, SAL_TLS_BUFSZ - 1, 0);
if (bytes_received <= 0)
{
    rt_kprintf("received error,close the socket.\n");
    goto __exit;
}

rt_kprintf("recv data:\n");
for (i = 0; i < bytes_received; i++)
{
    rt_kprintf("%c", recv_data[i]);
}

__exit:
    if (recv_data)
        rt_free(recv_data);

    if (sock >= 0)
        closesocket(sock);
}

#endif /* FINSH_USING_MSH */
#include <finsh.h>
MSH_CMD_EXPORT(sal_tls_test, SAL TLS function test);
#endif /* FINSH_USING_MSH */
```

25.1.3 配置选项

当我们使用 SAL 组件时需要在 `rtconfig.h` 中定义如下宏定义：

宏定义	描述
<code>RT_USING_SAL</code>	开启 SAL 功能
<code>SAL_USING_LWIP</code>	开启 lwIP 协议栈支持
<code>SAL_USING_AT</code>	开启 AT Socket 协议栈支持
<code>SAL_USING_TLS</code>	开启 SAL TLS 功能支持
<code>SAL_USING_POSIX</code>	开启 POSIX 文件系统相关函数支持，如 <code>read</code> 、 <code>write</code> 、 <code>select/poll</code> 等

目前 SAL 抽象层支持 lwIP 协议栈、AT Socket 协议栈和 WIZnet 硬件 TCP/IP 协议栈，系统中开启 SAL 需要至少开启一种协议栈支持。

上面配置选项可以直接在 `rtconfig.h` 文件中添加使用，也可以通过组件包管理工具 ENV 配置选项加入，ENV 工具中具体配置路径如下：

```
RT-Thread Components --->
  Network --->
    Socket abstraction layer --->
      [*] Enable socket abstraction layer
          protocol stack implement --->
            [ ] Support lwIP stack
            [ ] Support AT Commands stack
            [ ] Support MbedTLS protocol
      [*]   Enable BSD socket operated by file system API
```

配置完成可以通过 `scons` 命令重新生成功能，完成 SAL 组件的添加。

25.2 初始化

配置开启 SAL 选项之后，需要在启动时对它进行初始化，开启 SAL 功能，如果程序中已经使用了组件自动初始化，则不再需要额外进行单独的初始化，否则需要在初始化任务中调用如下函数：

```
int sal_init(void);
```

该初始化函数主要是对 SAL 组件进行初始，支持组件重复初始化判断，完成对组件中使用的互斥锁等资源的初始化。SAL 组件中没有创建新的线程，这也意味着 SAL 组件资源占用极小，目前 **SAL** 组件资源占用为 **ROM 2.8K 和 RAM 0.6K**。

25.3 BSD Socket API 介绍

SAL 组件抽象出标准 BSD Socket API 接口，如下是对常用网络接口的介绍：

25.3.1 创建套接字 (socket)

```
int socket(int domain, int type, int protocol);
```

参数	描述
domain	协议族类型
type	协议类型
protocol	实际使用的运输层协议
返回	-
>=0	成功，返回一个代表套接字描述符的整数
-1	失败

该函数用于根据指定的地址族、数据类型和协议来分配一个套接字描述符及其所用的资源。

domain / 协议族类型:

- AF_INET: IPv4
- AF_INET6: IPv6

type / 协议类型:

- SOCK_STREAM: 流套接字
- SOCK_DGRAM: 数据报套接字
- SOCK_RAW: 原始套接字

25.3.2 绑定套接字 (bind)

```
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

参数	描述
s	套接字描述符
name	指向 sockaddr 结构体的指针，代表要绑定的地址
namelen	sockaddr 结构体的长度
返回	-
0	成功
-1	失败

该函数用于将端口号和 IP 地址绑定到指定套接字上。

SAL 组件依赖 netdev 组件，当使用 `bind()` 函数时，可以通过 netdev 网卡名称获取网卡对象中 IP 地址信息，用于将创建的 Socket 套接字绑定到指定的网卡对象。下面示例完成通过传入的网卡名称绑定该网卡 IP 地址并和服务器进行连接的过程：

```
#include <rtthread.h>
#include <arpa/inet.h>
#include <netdev.h>

#define SERVER_HOST    "192.168.1.123"
#define SERVER_PORT    1234

static int bing_test(int argc, char **argv)
{
    struct sockaddr_in client_addr;
    struct sockaddr_in server_addr;
    struct netdev *netdev = RT_NULL;
    int sockfd = -1;

    if (argc != 2)
    {
        rt_kprintf("bind_test [netdev_name] --bind network interface device by name
                   .\n");
        return -RT_ERROR;
    }

    /* 通过名称获取 netdev 网卡对象 */
    netdev = netdev_get_by_name(argv[1]);
    if (netdev == RT_NULL)
    {
        rt_kprintf("get network interface device(%s) failed.\n", argv[1]);
        return -RT_ERROR;
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        rt_kprintf("Socket create failed.\n");
        return -RT_ERROR;
    }

    /* 初始化需要绑定的客户端地址 */
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(8080);
    /* 获取网卡对象中 IP 地址信息 */
    client_addr.sin_addr.s_addr = netdev->ip_addr.addr;
    rt_memset(&(client_addr.sin_zero), 0, sizeof(client_addr.sin_zero));

    if (bind(sockfd, (struct sockaddr *)&client_addr, sizeof(struct sockaddr)) < 0)
    {
        rt_kprintf("socket bind failed.\n");
    }
}
```

```

        closesocket(sockfd);
        return -RT_ERROR;
    }
    rt_kprintf("socket bind network interface device(%s) success!\n", netdev->name);

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    server_addr.sin_addr.s_addr = inet_addr(SERVER_HOST);
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

    /* 连接到服务端 */
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) <
        0)
    {
        rt_kprintf("socket connect failed!\n");
        closesocket(sockfd);
        return -RT_ERROR;
    }
    else
    {
        rt_kprintf("socket connect success!\n");
    }

    /* 关闭连接 */
    closesocket(sockfd);
    return RT_EOK;
}

#endif FINSH_USING_MSH
#include <finsh.h>
MSH_CMD_EXPORT(bing_test, bind network interface device test);
#endif /* FINSH_USING_MSH */

```

25.3.3 监听套接字 (listen)

```
int listen(int s, int backlog);
```

参数	描述
s	套接字描述符
backlog	表示一次能够等待的最大连接数目
返回	-
0	成功
-1	失败

该函数用于 TCP 服务器监听指定套接字连接。

25.3.4 接收连接 (accept)

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

参数	描述
s	套接字描述符
addr	表示一次能够等待的最大连接数目
addrlen	客户端设备地址结构体的长度
返回	-
0	成功，返回新创建的套接字描述符
-1	失败

当应用程序监听来自其他主机的连接时，使用 `accept()` 函数初始化连接，`accept()` 为每个连接创立新的套接字并从监听队列中移除这个连接。

25.3.5 建立连接 (connect)

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

参数	描述
s	套接字描述符
name	服务器地址信息
namelen	服务器地址结构体的长度
返回	-
0	成功，返回新创建的套接字描述符
-1	失败

该函数用于建立与指定 socket 的连接。

25.3.6 TCP 数据发送 (send)

```
int send(int s, const void *dataptr, size_t size, int flags);
```

参数	描述
s	套接字描述符
dataptr	发送的数据指针
size	发送的数据长度
flags	标志，一般为 0
返回	-
>0	成功，返回发送的数据的长度
<=0	失败

该函数常用于 TCP 连接发送数据。

25.3.7 TCP 数据接收（recv）

```
int recv(int s, void *mem, size_t len, int flags);
```

参数	描述
s	套接字描述符
mem	接收的数据指针
len	接收的数据长度
flags	标志，一般为 0
返回	-
>0	成功，返回接收的数据的长度
=0	目标地址已传输完并关闭连接
<0	失败

该函数用于 TCP 连接接收数据。

25.3.8 UDP 数据发送（sendto）

```
int sendto(int s, const void *dataptr, size_t size, int flags, const struct sockaddr
          *to, socklen_t tolen);
```

参数	描述
s	套接字描述符
dataptr	发送的数据指针

参数	描述
size	发送的数据长度
flags	标志，一般为 0
to	目标地址结构体指针
tolen	目标地址结构体长度
返回	-
>0	成功，返回发送的数据的长度
<=0	失败

该函数用于 UDP 连接发送数据。

25.3.9 UDP 数据接收（recvfrom）

```
int recvfrom(int s, void *mem, size_t len, int flags, struct sockaddr *from,
            socklen_t *fromlen);
```

参数	描述
s	套接字描述符
mem	接收的数据指针
len	接收的数据长度
flags	标志，一般为 0
from	接收地址结构体指针
fromlen	接收地址结构体长度
返回	-
>0	成功，返回接收的数据的长度
=0	接收地址已传输完并关闭连接
<0	失败

该函数用于 UDP 连接接收数据。

25.3.10 关闭套接字（closesocket）

```
int closesocket(int s);
```

参数	描述
s	套接字描述符
返回	-
0	成功
-1	失败

该函数用于关闭连接，释放资源。

25.3.11 按设置关闭套接字（shutdown）

```
int shutdown(int s, int how);
```

参数	描述
s	套接字描述符
how	套接字控制的方式
返回	-
0	成功
-1	失败

该函数提供更多的权限控制套接字的关闭过程。

how / 套接字控制的方式：

- 0：停止接收当前数据，并拒绝以后的数据接收；
- 1：停止发送数据，并丢弃未发送的数据；
- 2：停止接收和发送数据。

25.3.12 设置套接字选项（setsockopt）

```
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

参数	描述
s	套接字描述符
level	协议栈配置选项
optname	需要设置的选项名
optval	设置选项值的缓冲区地址
optlen	设置选项值的缓冲区长度

参数	描述
返回	-
=0	成功
<0	失败

该函数用于设置套接字模式，修改套接字配置选项。

level / 协议栈配置选项：

- SOL_SOCKET: 套接字层
- IPPROTO_TCP: TCP 层
- IPPROTO_IP: IP 层

optname / 需要设置的选项名：

- SO_KEEPALIVE: 设置保持连接选项
- SO_RCVTIMEO: 设置套接字数据接收超时
- SO_SNDTIMEO: 设置套接数据发送超时

25.3.13 获取套接字选项 (getsockopt)

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
```

参数	描述
s	套接字描述符
level	协议栈配置选项
optname	需要设置的选项名
optval	获取选项值的缓冲区地址
optlen	获取选项值的缓冲区长度地址
返回	-
=0	成功
<0	失败

该函数用于获取套接字配置选项。

25.3.14 获取远端地址信息 (getpeername)

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

参数	描述
s	套接字描述符
name	接收信息的地址结构体指针
namelen	接收信息的地址结构体长度
返回	-
=0	成功
<0	失败

该函数用于获取与套接字相连的远端地址信息。

25.3.15 获取本地地址信息 (getsockname)

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

参数	描述
s	套接字描述符
name	接收信息的地址结构体指针
namelen	接收信息的地址结构体长度
返回	-
=0	成功
<0	失败

该函数用于获取本地套接字地址信息。

25.3.16 配置套接字参数 (ioctlsocket)

```
int ioctlsocket(int s, long cmd, void *arg);
```

参数	描述
s	套接字描述符
cmd	套接字操作命令
arg	操作命令所带参数
返回	-
=0	成功
<0	失败

该函数用于设置套接字控制模式。

cmd 支持下列命令

- **FIONBIO:** 开启或关闭套接字的非阻塞模式, **arg** 参数 1 为开启非阻塞, 0 为关闭非阻塞。

25.4 网络协议栈接入方式

网络协议栈或网络功能实现的接入, 主要是对协议簇结构体的初始化和注册处理, 并且添加到 SAL 组件中协议簇列表中, 协议簇结构体定义如下:

```
/* network interface socket operations */
struct sal_socket_ops
{
    int (*socket)      (int domain, int type, int protocol);
    int (*closesocket)(int s);
    int (*bind)        (int s, const struct sockaddr *name, socklen_t namelen);
    int (*listen)      (int s, int backlog);
    int (*connect)     (int s, const struct sockaddr *name, socklen_t namelen);
    int (*accept)      (int s, struct sockaddr *addr, socklen_t *addrlen);
    int (*sendto)      (int s, const void *data, size_t size, int flags, const struct
                        sockaddr *to, socklen_t tolen);
    int (*recvfrom)    (int s, void *mem, size_t len, int flags, struct sockaddr *
                        from, socklen_t *fromlen);
    int (*getsockopt)  (int s, int level, int optname, void *optval, socklen_t *
                        optlen);
    int (*setsockopt)  (int s, int level, int optname, const void *optval, socklen_t *
                        optlen);
    int (*shutdown)    (int s, int how);
    int (*getpeername)(int s, struct sockaddr *name, socklen_t *namelen);
    int (*getsockname)(int s, struct sockaddr *name, socklen_t *namelen);
    int (*ioctlsocket)(int s, long cmd, void *arg);
#define SAL_USING_POSIX
    int (*poll)        (struct dfs_fd *file, struct rt_pollreq *req);
#endif
};

/* sal network database name resolving */
struct sal_netdb_ops
{
    struct hostent* (*gethostbyname) (const char *name);
    int             (*gethostbyname_r)(const char *name, struct hostent *ret, char *
                                         buf, size_t buflen, struct hostent **result, int *h_errnop);
    int             (*getaddrinfo)   (const char *nodename, const char *servname,
                                         const struct addrinfo *hints, struct addrinfo **res);
    void            (*freeaddrinfo) (struct addrinfo *ai);
};

/* 协议簇结构体定义 */
```

```

struct sal_proto_family
{
    int family;                                /* primary protocol families type
    */
    int sec_family;                             /* secondary protocol families type
    */
    const struct sal_socket_ops *skt_ops;       /* socket opreations */
    const struct sal_netdb_ops *netdb_ops;       /* network database opreations */
};

```

- **family**: 每个协议栈支持的主协议簇类型，例如 lwIP 的为 AF_INET，AT Socket 为 AF_AT，WIZnet 为 AF_WIZ。
- **sec_family**: 每个协议栈支持的次协议簇类型，用于支持单个协议栈或网络实现时，匹配软件包中其他类型的协议簇类型。
- **skt_ops**: 定义 socket 相关执行函数，如 connect、send、recv 等，每种协议簇都有一组不同的实现方式。
- **netdb_ops**: 定义非 socket 相关执行函数，如 gethostbyname、getaddrinfo、freeaddrinfo 等，每种协议簇都有一组不同的实现方式。

以下为 AT Socket 网络实现的接入注册流程，开发者可参考实现其他的协议栈或网络实现的接入：

```

#include <rtthread.h>
#include <netdb.h>
#include <sal.h>          /* SAL 组件结构体存放头文件 */
#include <at_socket.h>      /* AT Socket 相关头文件 */
#include <af_inet.h>

#include <netdev.h>        /* 网卡功能相关头文件 */

#ifndef SAL_USING_POSIX
#include <dfs_poll.h>      /* poll 函数实现相关头文件 */
#endif

#ifndef SAL_USING_AT
/* 自定义的 poll 执行函数，用于 poll 中处理接收的事件 */
static int at_poll(struct dfs_fd *file, struct rt_pollreq *req)
{
    int mask = 0;
    struct at_socket *sock;
    struct socket *sal_sock;

    sal_sock = sal_get_socket((int) file->data);
    if(!sal_sock)
    {
        return -1;
    }
}

```

```
sock = at_get_socket((int)sal_sock->user_data);
if (sock != NULL)
{
    rt_base_t level;

    rt_poll_add(&sock->wait_head, req);

    level = rt_hw_interrupt_disable();
    if (sock->rcvevent)
    {
        mask |= POLLIN;
    }
    if (sock->sendevent)
    {
        mask |= POLLOUT;
    }
    if (sock->errevent)
    {
        mask |= POLLERR;
    }
    rt_hw_interrupt_enable(level);
}

return mask;
}
#endif

/* 定义和赋值 Socket 执行函数, SAL 组件执行相关函数时调用该注册的底层函数 */
static const struct proto_ops at_inet_stream_ops =
{
    at_socket,
    at_closesocket,
    at_bind,
    NULL,
    at_connect,
    NULL,
    at_sendto,
    at_recvfrom,
    at_getsockopt,
    at_setsockopt,
    at_shutdown,
    NULL,
    NULL,
    NULL,
    #ifdef SAL_USING_POSIX
    at_poll,
    #else
    NULL,

```

```
#endif /* SAL_USING_POSIX */  
};  
  
static const struct sal_netdb_ops at_netdb_ops =  
{  
    at_gethostbyname,  
    NULL,  
    at_getaddrinfo,  
    at_freeaddrinfo,  
};  
  
/* 定义和赋值 AT Socket 协议簇结构体 */  
static const struct sal_proto_family at_inet_family =  
{  
    AF_AT,  
    AF_INET,  
    &at_socket_ops,  
    &at_netdb_ops,  
};  
  
/* 用于设置网卡设备中协议簇相关信息 */  
int sal_at_netdev_set_pf_info(struct netdev *netdev)  
{  
    RT_ASSERT(netdev);  
  
    netdev->sal_user_data = (void *) &at_inet_family;  
    return 0;  
}  
  
#endif /* SAL_USING_AT */
```

第 26 章

ulog 日志

26.1 ulog 简介

日志的定义: 日志是将软件运行的状态、过程等信息，输出到不同的介质中（例如：文件、控制台、显示屏等），并进行显示和保存。为软件调试、维护过程中问题追溯、性能分析、系统监控、故障预警等功能，提供参考依据。可以说，日志的使用，几乎占用的软件生命周期的至少 80% 的时间。

日志的重要性: 对于操作系统而言，由于其软件的复杂度非常大，单步调试在一些场景下并不适合，所以日志组件在操作系统上几乎都是标配。完善的日志系统也能让操作系统的调试事半功倍。

ulog 的起源: RT-Thread 一直缺少小巧、实用的日志组件，而 ulog 的诞生补全了这块的短板。它将作为 RT-Thread 的基础组件被开源出来，让我们的开发者也能用上简洁易用的日志系统，提高开发效率。

ulog 是一个非常简洁、易用的 C/C++ 日志组件，第一个字母 u 代表 μ ，即微型的意思。它能做到最低 **ROM<1K, RAM<0.2K** 的资源占用。ulog 不仅有小巧体积，同样也有非常全面的功能，其设计理念参考的是另外一款 C/C++ 开源日志库：EasyLogger（简称 elog），并在功能和性能等方面做了非常多的改进。主要特性如下：

- 日志输出的后端多样化，可支持例如：串口、网络，文件、闪存等后端形式。
- 日志输出被设计为线程安全的方式，并支持异步输出模式。
- 日志系统高可靠，在中断 ISR、Hardfault 等复杂环境下依旧可用。
- 日志支持运行期 / 编译期设置输出级别。
- 日志内容支持按关键词及标签方式进行全局过滤。
- API 和日志格式可兼容 linux syslog。
- 支持以 hex 格式 dump 调试数据到日志中。
- 兼容 rtdbg（RTT 早期的日志头文件）及 EasyLogger 的日志输出 API。

26.1.1 ulog 架构

下图为 ulog 日志组件架构图：

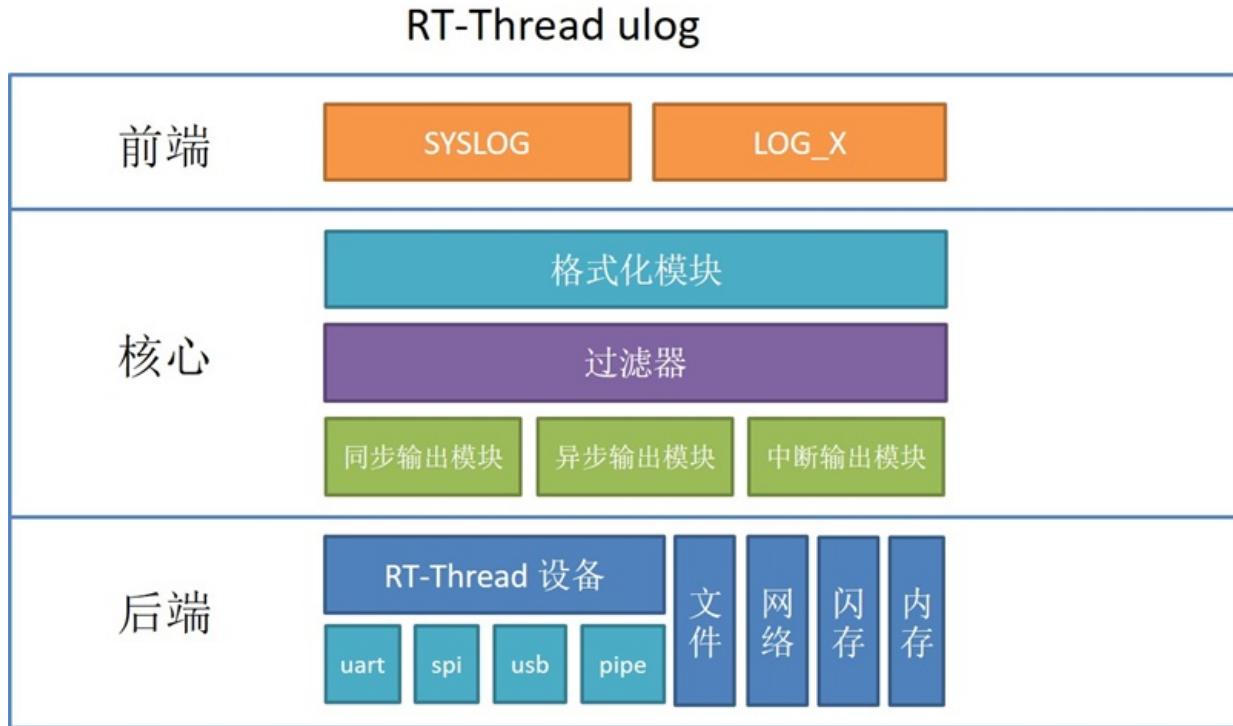


图 26.1: ulog 框架

- **前端**: 该层作为离应用最近的一层，给用户提供了 syslog 及 LOG_X 两类 API 接口，方便用户在不同的场景中使用。
- **核心**: 中间核心层的主要工作是将上层传递过来的日志，按照不同的配置要求进行格式化与过滤然后生成日志帧，最终通过不同的输出模块，输出到最底层的后端设备上。
- **后端**: 接收到核心层发来的日志帧后，将日志输出到已经注册的日志后端设备上，例如：文件、控制台、日志服务器等等。

26.1.2 配置选项

ENV 工具中使用 menuconfig 配置 ulog 的路径如下所示：

RT-Thread Components □ Utilities □ Enable ulog

ulog 配置选项说明如下所示，一般情况下使用默认配置即可：

```
[*] Enable ulog          /* 使能 ulog */
    The static output log level./* 选择静态的日志输出级别。选择完成后，比设定级别
        低的日志（这里特指使用 LOG_X API 的日志）将不会被编译到 ROM 中 */
[ ]  Enable ISR log.     /* 使能中断 ISR 日志，即在 ISR 中也可以使用日志输
    出 API */
```

```
[*] Enable assert check.          /* 使能断言检查。关闭后，断言的日志将不会被编译到
ROM 中 */
(128) The log's max width.      /* 日志的最大长度。由于 ulog 的日志 API 按行作为单
位，所以这个长度也代表一行日志的最大长度 */
[ ] Enable async output mode.   /* 使能异步日志输出模式。开启这个模式后，日志不会
立刻输出到后端，而是先缓存起来，然后交给日志输出线程（例如：idle 线程）去输出 */
    log format --->           /* 配置日志的格式，例如：时间信息，颜色信息，线程
信息，是否支持浮点等等 */
[*] Enable console backend.     /* 使能控制台作为后端。使能后日志可以输出到控制台
串口上。建议保持开启。 */
[ ] Enable runtime log filter.  /* 使能运行时的日志过滤器，即动态过滤。使能后，日
志将支持按标签、关键词等方式，在系统运行时进行动态过滤。 */
```

配置日志的格式（**log format**）选项描述如下所示：

```
[ ] Enable float number support. It will using more thread stack. /* 浮点型数字的
支持（传统的 rt_dbg/rt_kprintf 均不支持浮点数日志） */
[*] Enable color log.           /* 带颜色的日志 */
[*] Enable time information.    /* 时间信息 */
[ ] Enable timestamp format for time. /* 包括时间戳 */
[*] Enable level information.   /* 级别信息 */
[*] Enable tag information.     /* 标签信息 */
[ ] Enable thread information.  /* 线程信息 */
```

26.1.3 日志级别

日志级别代表了日志的重要性，在 ulog 中由高到低，有如下几个日志级别：

级别	名称	描述
LOG_LVL_ASSERT	断言	发生无法处理、致命性的错误，以至于系统无法继续运行的断言日志
LOG_LVL_ERROR	错误	发生严重的、不可修复的错误时输出的日志属于错误级别日志
LOG_LVL_WARNING	警告	出现一些不太重要的、具有可修复性的错误时，会输出这些警告日志
LOG_LVL_INFO	信息	给本模块上层使用人员查看的重要提示信息日志，例如：初始化成功，当前工作状态等。该级别日志一般在量产时依旧保留
LOG_LVL_DBG	调试	给本模块开发人员查看的调试日志，该级别日志一般在量产时关闭

在 ulog 中日志级别还有如下分类：

- 静态级别与动态级别：按照日志是否可以在运行阶段修改进行分类。可在运行阶段修改的称之为动态

级别，只能在编译阶段修改的称之为静态级别。比静态级别低的日志（这里特指使用 LOG_X API 的日志）将不会被编译到 ROM 中，最终也不会输出、显示出来。而动态级别可以管控的是高于或等于静态级别的日志。在 ulog 运行时，比动态级别低的日志会被过滤掉。

- **全局级别与模块级别：**按照作用域进行的分类。在 ulog 中每个文件（模块）也可以设定独立的日志级别。全局级别作用域大于模块级别，也就是模块级别只能管控那些高于或等于全局级别的模块日志。

综合上面分类可以看出，在 ulog 可以通过以下 4 个方面来设定日志的输出级别：

- 全局静态日志级别：在 menuconfig 中配置，对应 ULOG_OUTPUT_LVL 宏。
- 全局动态日志级别：使用 void ulog_global_filter_lvl_set(rt_uint32_t level) 函数来设定。
- 模块静态日志级别：在模块（文件）内定义 LOG_LVL 宏，与日志标签宏 LOG_TAG 定义方式类似。
- 模块动态日志级别：使用 int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level) 函数来设定。

它们的作用范围关系为：全局静态 > 全局动态 > 模块静态 > 模块动态。

26.1.4 日志标签

由于日志输出量的不断增大，为了避免日志被杂乱无章的输出出来，就需要使用标签（tag）给每条日志进行分类。标签的定义是按照模块化的方式，例如：Wi-Fi 组件包括设备驱动（wifi_driver）、设备管理（wifi_mgnt）等模块，则 Wi-Fi 组件内部模块可以使用 wifi.driver、wifi.mgnt 等作为标签，进行日志的分类输出。

每条日志的标签属性也可以被输出并显示出来，同时 ulog 还可以设置每个标签（模块）对应日志的输出级别，当前不重要模块的日志可以选择性关闭，不仅降低 ROM 资源，还能帮助开发者过滤无关日志。

参见 `rt-thread\examples\ulog_example.c` ulog 例程文件，在文件顶部有定义 LOG_TAG 宏：

```
#define LOG_TAG      "example"      // 该模块对应的标签。不定义时，默认：NO_TAG
#define LOG_LVL       LOG_LVL_DBG    // 该模块对应的日志输出级别。不定义时，默认：调试
级别
#include <ulog.h>                  // 必须在 LOG_TAG 与 LOG_LVL 下面
```

需要注意的，定义日志标签必须位于 `##include <ulog.h>` 的上方，否则会使用默认的 NO_TAG（不推荐定义在头文件中定义这些宏）。

日志标签的作用域是当前源码文件，项目源代码通常也会按照模块进行文件分类。所以在定义标签时，可以指定模块名、子模块名作为标签名称，这样不仅在日志输出显示时清晰直观，也能方便后续按标签方式动态调整级别或过滤。

26.2 日志初始化

26.2.1 初始化

```
int ulog_init(void)
```

返回	描述
>=0	成功
-5	失败，内存不足

在使用 `ulog` 前必须调用该函数完成 `ulog` 初始化。如果开启了组件自动初始化，该函数也将被自动调用。

26.2.2 去初始化

```
void ulog_deinit(void)
```

当 `ulog` 不再使用时，可以执行该 `deinit` 释放资源。

26.3 日志输出 API

`ulog` 主要有两种日志输出宏 API，源代码中定义如下所示：

<code>#define LOG_E(...)</code>	<code>ulog_e(LOG_TAG, __VA_ARGS__)</code>
<code>#define LOG_W(...)</code>	<code>ulog_w(LOG_TAG, __VA_ARGS__)</code>
<code>#define LOG_I(...)</code>	<code>ulog_i(LOG_TAG, __VA_ARGS__)</code>
<code>#define LOG_D(...)</code>	<code>ulog_d(LOG_TAG, __VA_ARGS__)</code>
<code>#define LOG_RAW(...)</code>	<code>ulog_raw(__VA_ARGS__)</code>
<code>#define LOG_HEX(name, width, buf, size)</code>	<code>ulog_hex(name, width, buf, size)</code>

- 宏 `LOG_X(...)`: `X` 对应的是不同级别的第一个字母大写。参数 `...` 为日志内容，格式与 `printf` 一致。这种方式是首选，一方面因为其 API 格式简单，入参只有一个即日志信息，再者还支持按模块静态日志级别过滤。
- 宏 `ulog_x(LOG_TAG, __VA_ARGS__)`: `x` 对应的是不同级别的简写。参数 `LOG_TAG` 为日志标签，参数 `...` 为日志内容，格式与 `printf` 一致。这个 API 适用于在一个文件中使用不同 tag 输出日志的情况。

API	描述
<code>LOG_E(...)</code>	错误级别日志
<code>LOG_W(...)</code>	错误级别日志
<code>LOG_I(...)</code>	提示级别日志
<code>LOG_D(...)</code>	调试级别日志
<code>LOG_RAW(...)</code>	输出 raw 日志
<code>LOG_HEX(name, width, buf, size)</code>	输出 16 进制格式数据到日志

`LOG_X` 及 `ulog_x` 这类 API 输出都是带格式日志，有些时候需要输出不带任何格式的日志时，可以使用 `LOG_RAW` 或 `ulog_raw()`。例如：

```
LOG_RAW("\r");
ulog_raw("\033[2A");
```

以 16 进制 hex 格式 dump 数据到日志中可使用可以使用 `LOG_HEX()` 或 `ulog_hex`。函数参数及描述如下所示：

参数	描述
<code>tag</code>	日志标签
<code>width</code>	一行 hex 内容的宽度（数量）
<code>buf</code>	待输出的数据内容
<code>size</code>	数据大小

hexdump 日志为 DEBUG 级别，支持运行期的级别过滤，hexdump 日志对应的 tag，支持运行期的标签过滤。

ulog 也提供里断言 API：`ASSERT(表达式)`，当断言触发时，系统会停止运行，内部也会执行 `ulog_flush()`，所有日志后端将执行 flush。如果开启了异步模式，缓冲区中所有的日志也将被 flush。断言的使用示例如下：

```
void show_string(const char *str)
{
    ASSERT(str);
    ...
}
```

26.4 日志使用示例

26.4.1 使用示例

下面将以 ulog 例程进行介绍，打开 `rt-thread\examples\ulog_example.c` 可以看到，顶部有定义该文件的标签及静态优先级。

```
#define LOG_TAG          "example"
#define LOG_LVL           LOG_LVL_DBG
#include <ulog.h>
```

在 `void ulog_example(void)` 函数中有使用 `LOG_X` API，大致如下：

```
/* output different level log by LOG_X API */
LOG_D("LOG_D(%d): RT-Thread is an open source IoT operating system from China.",
      count);
```

```

LOG_I("LOG_I(%d): RT-Thread is an open source IoT operating system from China.",  

      count);  

LOG_W("LOG_W(%d): RT-Thread is an open source IoT operating system from China.",  

      count);  

LOG_E("LOG_E(%d): RT-Thread is an open source IoT operating system from China.",  

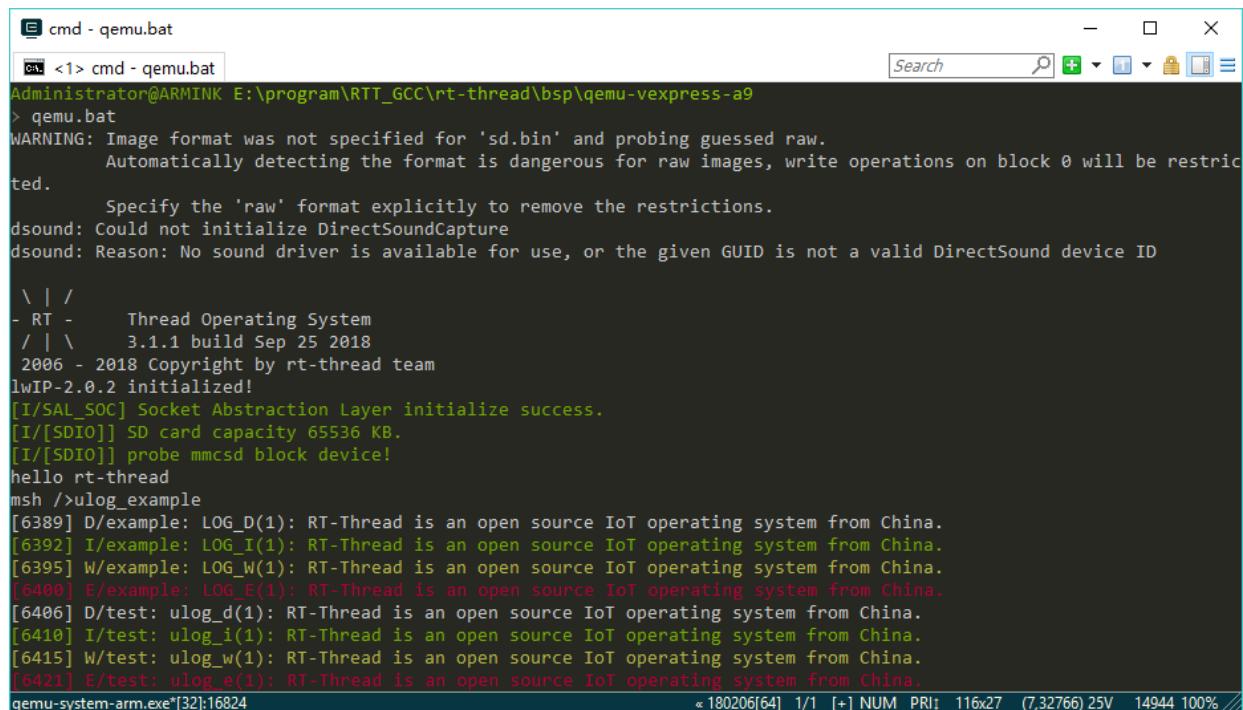
      count);

```

这些日志输出 API 均支持 `printf` 格式，并且会在日志末尾自动换行。

下面将在 `qemu` 上展示下 `ulog` 例程的运行效果：

- 将 `rt-thread\examples\ulog_example.c` 拷贝至 `rt-thread\bsp\qemu-vexpress-a9\applications` 文件夹下
- 在 Env 中进入 `rt-thread\bsp\qemu-vexpress-a9` 目录
- 确定之前已执行过 `ulog` 的配置后，执行 `scons` 命令并等待编译完成
- 运行 `qemu.bat` 来打开 RT-Thread 的 `qemu` 模拟器
- 输入 `ulog_example` 命令，即可看到 `ulog` 例程运行结果，大致效果如下图



```

Administrator@ARMINK E:\program\RTT_GCC\rt-thread\bsp\qemu-vexpress-a9
> qemu.bat
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
          Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
          Specify the 'raw' format explicitly to remove the restrictions.
dsound: Could not initialize DirectSoundCapture
dsound: Reason: No sound driver is available for use, or the given GUID is not a valid DirectSound device ID

\ | /
- RT - Thread Operating System
/ | \ 3.1.1 build Sep 25 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh />ulog_example
[6389] D/example: LOG_D(1): RT-Thread is an open source IoT operating system from China.
[6392] I/example: LOG_I(1): RT-Thread is an open source IoT operating system from China.
[6395] W/example: LOG_W(1): RT-Thread is an open source IoT operating system from China.
[6400] E/example: LOG_E(1): RT-Thread is an open source IoT operating system from China.
[6406] D/test: ulog_d(1): RT-Thread is an open source IoT operating system from China.
[6410] I/test: ulog_i(1): RT-Thread is an open source IoT operating system from China.
[6415] W/test: ulog_w(1): RT-Thread is an open source IoT operating system from China.
[6421] E/test: ulog_e(1): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe*32]:16824  * 180206[64] 1/1 [+]

```

图 26.2: `ulog` 例程

可以看到每条日志都是按行显示，不同级别日志也有着不同的颜色。在日志最前面有当前系统的 tick，中间有显示日志级别及标签，最后面是具体日志内容。在本文后面也会重点介绍这些日志格式及配置说明。

26.4.2 在中断 ISR 中使用

很多时候需要在中断 ISR 中输出日志，但是中断 ISR 可能会打断正在进行日志输出的线程。要保证中断日志与线程日志互不干涉，就得针对于中断情况进行特殊处理。

ulog 已集成中断日志的功能，但是默认没有开启，使用时打开 `Enable ISR log` 选项即可，日志的 API 与线程中使用的方式一致，例如：

```
#define LOG_TAG          "driver.timer"
#define LOG_LVL           LOG_LVL_DBG
#include <ulog.h>

void Timer2_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    LOG_D("I'm in timer2 ISR");

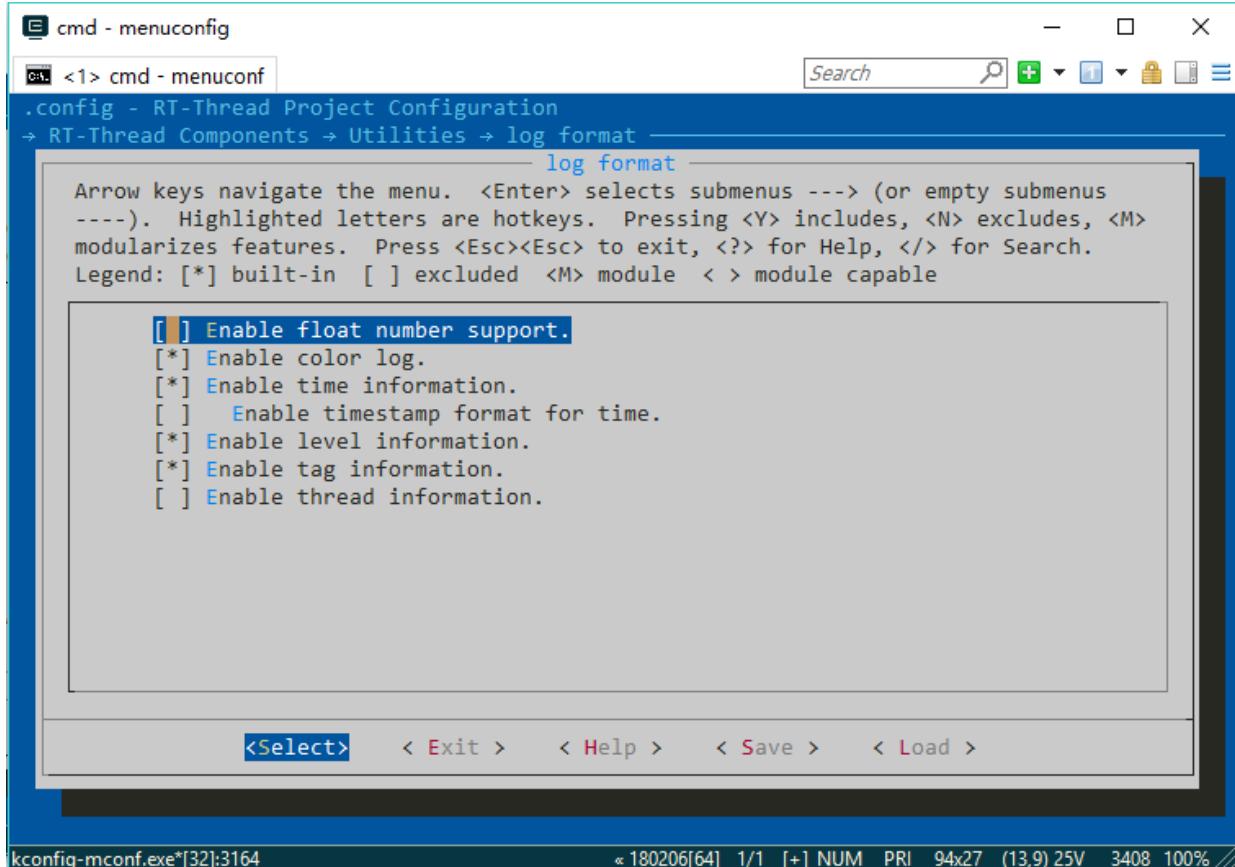
    /* leave interrupt */
    rt_interrupt_leave();
}
```

这里说明下中断日志在 ulog 处于同步模式与异步模式下的不同策略：

-**同步模式下**：如果线程此时正在输出日志时来了中断，此时如果中断里也有日志要输出，会直接输出到控制台上，不支持输出到其他后端；-**异步模式下**：如果发生上面的情况，中断里的日志会先放入缓冲区中，最终和线程日志一起交给日志输出线程来处理。

26.4.3 设置日志格式

ulog 支持的日志格式可以在 `menuconfig` 中配置，位于 `RT-Thread Components` \square `Utilities` \square `ulog` \square `log format`，具体配置如下：

图 26.3: *ulog* 格式配置

分别可以配置：浮点型数字的支持（传统的 `rtdbg/rt_kprintf` 均不支持浮点数日志）、带颜色的日志、时间信息（包括时间戳）、级别信息、标签信息、线程信息。下面我们将这些选项全部选中，保存后重新编译并在 `qemu` 中再次运行 `ulog` 例程，看下实际的效果：

```

cmd - qemu.bat
Administrator@ARMINK E:\program\RTT_GCC\rt-thread\bsp\qemu-vexpress-a9
> qemu.bat
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
          Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
          Specify the 'raw' format explicitly to remove the restrictions.
dsound: Could not initialize DirectSoundCapture
dsound: Reason: No sound driver is available for use, or the given GUID is not a valid DirectSound device ID

\ | /
- RT - Thread Operating System
/ | \
  3.1.1 build Sep 26 2018
  2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh >ulog_example
01-01 00:00:34.463 D/example tshell: LOG_D(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.468 I/example tshell: LOG_I(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.472 W/example tshell: LOG_W(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.476 E/example tshell: LOG_E(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.480 D/test tshell: ulog_d(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.485 I/test tshell: ulog_i(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.489 W/test tshell: ulog_w(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:34.493 E/test tshell: ulog_e(1): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe*32:12156 * 180206[64] 1/1 [+ NUM PRI: 113x27 (7,32766) 25V 3408 100%

```

图 26.4: ulog 例程(全部格式)

可以看出，相比第一次运行例程，时间信息已经由系统的 tick 数值变为时间戳信息，并且线程信息也被输出出来。

26.4.4 hexdump 输出使用

hexdump 也是日志输出时较为常用的功能，通过 hexdump 可以将一段数据以 hex 格式输出出来，对应的 API 为：`void ulog_hexdump(const char *tag, rt_size_t width, rt_uint8_t *buf, rt_size_t size)`，下面看下具体的使用方法及运行效果：

```

/* 定义一个 128 个字节长度的数组 */
uint8_t i, buf[128];
/* 在数组内填充上数字 */
for (i = 0; i < sizeof(buf); i++)
{
    buf[i] = i;
}
/* 以 hex 格式 dump 数组内的数据，宽度为 16 */
ulog_hexdump("buf_dump_test", 16, buf, sizeof(buf));

```

可以将上面的代码拷贝到 ulog 例程中运行，然后再看下实际运行结果：

```

\\ | /
- RT - Thread Operating System
/ | \ 3.1.1 build Sep 26 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk block device!
hello rt-thread
msh >ulog example
D/HEX buf_dump_test: 0000-0010: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ..... .
D/HEX buf_dump_test: 0010-0020: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F ..... .
D/HEX buf_dump_test: 0020-0030: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#$%&'()*+,.-./
D/HEX buf_dump_test: 0030-0040: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 0123456789:;<->?
D/HEX buf_dump_test: 0040-0050: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F @ABCDEFGHIJKLMNO
D/HEX buf_dump_test: 0050-0060: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F PQRSTUWVXYZ[\]^_
D/HEX buf_dump_test: 0060-0070: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F `abcdefghijklmn
D/HEX buf_dump_test: 0070-0080: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F pqrstuvwxyz{|}~.

01-01 00:00:31.107 D/example tshell: LOG_D(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.112 I/example tshell: LOG_I(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.117 W/example tshell: LOG_W(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.122 E/example tshell: LOG_E(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.126 D/test tshell: ulog_d(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.131 I/test tshell: ulog_i(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.135 W/test tshell: ulog_w(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.140 E/test tshell: ulog_e(1): RT-Thread is an open source IoT operating system from China.

qemu-system-arm.exe*[32]:13824

```

图 26.5: ulog 例程 (hexdump)

可以看出，中部为 buf 数据的 16 进制 hex 信息，最右侧为各个数据对应的字符信息。

26.5 日志高级功能

在了解了前面小节对日志的介绍，`ulog` 的基本功能都可以掌握了。为了让大家更好的玩转 `ulog`，这篇应用笔记会重点跟大家介绍 `ulog` 的高级功能及一些日志调试的经验和技巧。学会这些高级用法以后，开发者也能很大程度上提升日志调试的效率。

同时还会介绍 `ulog` 的高级模式：`syslog` 模式，这个模式能做到从前端 API 到日志格式对于 Linux `syslog` 的完全兼容，极大的方便从 Linux 上的迁移过来的软件。

26.5.1 日志后端

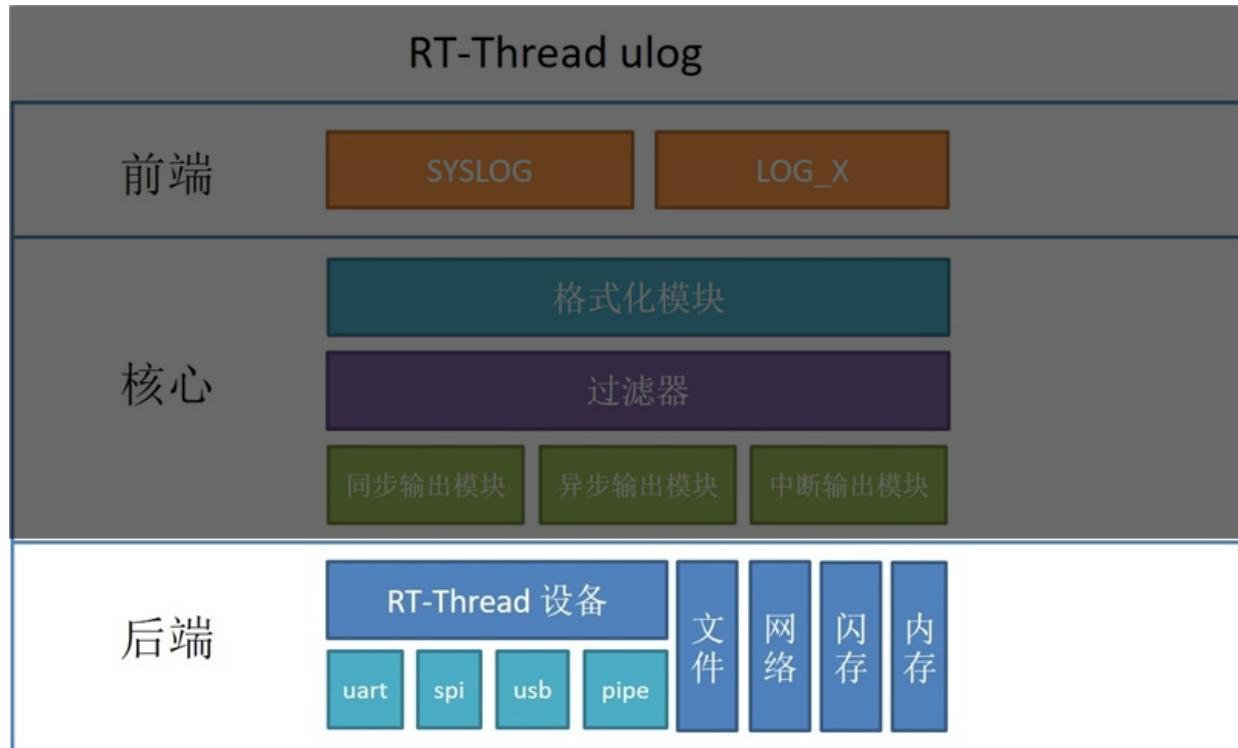


图 26.6: *uLog* 框架

讲到后端，我们来回顾下 *uLog* 的框架图。通过上图可以看出，*uLog* 是采用前后端分离的设计，前端与后端无依赖。并且支持的后端多样化，无论什么样后端，只要实现出来，都可以注册上去。

目前 *uLog* 已集成控制台后端，即传统的输出 `rt_kprintf` 打印日志的设备。*uLog* 还支持 Flash 后端，与 EasyFlash 无缝结合，详见其软件包（[点击查看](#)）。后期 *uLog* 还会增加文件后端、网络后端等后端的实现。当然，如果有特殊需求，用户也可以自己来实现后端。

26.5.1.1 注册后端设备

```
rt_err_t ulog_backend_register(ulog_backend_t backend, const char *name, rt_bool_t support_color)
```

参数	描述
backend	要注册的后端设备句柄
name	后端设备名称
support_color	是否支持彩色日志
返回	-
>=0	成功

该函数用于将后端设备注册到 uLog 中，注册前确保后端设备结构体中的函数成员已设置。

26.5.1.2 注销后端设备

```
rt_err_t ulog_backend_unregister(ulog_backend_t backend);
```

参数	描述
backend	要注销的后端设备句柄
返回	-
>=0	成功

该函数用于注销已经注册的后端设备。

26.5.1.3 后端实现及注册示例

下面以控制台后端为例，简单介绍后端的实现方法及注册方法。

打开 `rt-thread/components/utilities/ulog/backend/console_be.c` 文件，可以看到大致有如下内容：

```
#include <rthw.h>
#include <ulog.h>

/* 定义控制台后端设备 */
static struct ulog_backend console;
/* 控制台后端输出函数 */
void ulog_console_backend_output(struct ulog_backend *backend, rt_uint32_t level,
    const char *tag, rt_bool_t is_raw, const char *log, size_t len)
{
    ...
    /* 输出日志到控制台 */
    ...
}

/* 控制台后端初始化 */
int ulog_console_backend_init(void)
{
    /* 设定输出函数 */
    console.output = ulog_console_backend_output;
    /* 注册后端 */
    ulog_backend_register(&console, "console", RT_TRUE);

    return 0;
}
INIT_COMPONENT_EXPORT(ulog_console_backend_init);
```

通过上面的代码可以看出控制台后端的实现非常简单，这里实现了后端设备的 `output` 函数，并将该后端注册到 `ulog` 里，之后 `ulog` 的日志都会输出到控制台上。

如果要实现一个比较复杂的后端设备，此时就需要了解后端设备结构体，具体如下：

```
struct ulog_backend
{
    char name[RT_NAME_MAX];
    rt_bool_t support_color;
    void (*init) (struct ulog_backend *backend);
    void (*output)(struct ulog_backend *backend, rt_uint32_t level, const char *tag,
                  rt_bool_t is_raw, const char *log, size_t len);
    void (*flush) (struct ulog_backend *backend);
    void (*deinit)(struct ulog_backend *backend);
    rt_slist_t list;
};
```

从这个结构体的角度可以看出，实现后端设备的要求如下：

- `name` 以及 `support_color` 属性可以通过 `ulog_backend_register` 函数在注册时传入。
- `output` 为后端具体的输出函数，所有后端都必须实现接口。
- `init/deinit` 可选择性实现，`init` 会在 `register` 时调用，`deinit` 会在 `ulog_deinit` 时调用。
- `flush` 也是可选择性实现，一些内部输出带缓存的后端需要必须实现该接口。比如一些带 RAM 缓存的文件系统。后端的 `flush` 一般会在断言、`hardfault` 等异常情况下由 `ulog_flush` 完成调用。

26.5.2 异步日志

在 `ulog` 中，默认的输出模式是同步模式，在很多场景下用户可能还需要异步模式。用户在调用日志输出 API 时，会将日志缓存到缓冲区中，会有专门负责日志输出的线程取出日志，然后输出到后端。

异步模式和同步模式针对用户而言，在日志 API 使用上是没有差异的，因为 `ulog` 在底层处理上会有区分。两者的工作原理区别大致如下图所示：

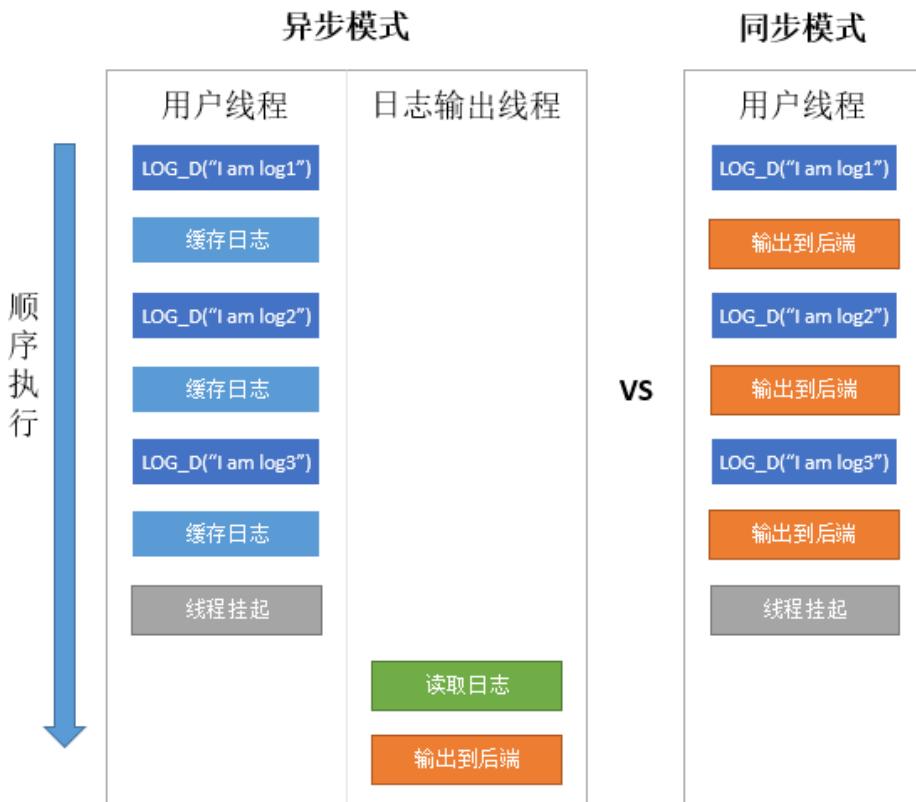


图 26.7: ulog 异步 VS 同步

异步模式的优缺点如下：

优点:

- 首先日志输出时不会阻塞住当前线程，再加上有些后端输出速率低，所以使用同步输出模式可能影响当前线程的时序，异步模式不存在该问题。
- 其次，由于每个使用日志的线程省略了后端输出的动作，所以这些线程的堆栈开销可能也会减少，从这个角度也可以降低整个系统的资源占用。
- 同步模式下的中断日志只能输出到控制台后端，而异步模式下中断日志可以输出到所有后端去。

缺点：首先异步模式需要日志缓冲区。再者异步日志的输出还需要有专门线程来完成，比如：idle 线程或者用户自定义的线程，用法上略显复杂。整体感觉异步模式资源占用会比同步模式要高。

26.5.2.1 配置选项

在 ENV 工具中使用 menuconfig 进入 ulog 配置选项：

RT-Thread Components □ Utilities □ Enable ulog

异步模式相关配置选项描述如下所示：

<code>[*] Enable async output mode.</code>	<code>/* 使能异步模式 */</code>
<code>(2048) The async output buffer size.</code>	<code>/* 异步缓冲区大小，默认为 2048 */</code>

```
[*]      Enable async output by thread.          /* 是否启用 ulog 里异步日志输出线
程，该线程运行时将会等待日志通知，然后输出日志到所有的后端。该选项默认开启，如果
想要修改为其他线程，例如：idle 线程，可关闭此选项。 */
(1024)    The async output thread stack size.    /* 异步输出线程的堆栈大小，默认为
1024 */
(30)      The async output thread stack priority./* 异步输出线程的优先级，默认为 30
*/
*/
```

使用 `idle` 线程输出时，实现虽然很简单，只需在应用层调用 `rt_thread_idle_sethook(ulog_async_output)` 即可，但也会存在一些限制。

- `idle` 线程堆栈大小需要根据实际的后端使用情况进行调整。
- 由于在 `idle` 线程内部不允许执行线程挂起操作，所以 Flash、网络等后端可能无法基于 `idle` 线程使用。

26.5.2.2 使用示例

保存异步输出选项配置，将 `rt-thread\examples\ulog_example.c` 拷贝至 `rt-thread\bsp\qemu-vexpress-a9\applications` 文件夹下。

执行 `scons` 命令并等待编译完成。运行 `qemu.bat` 来打开 RT-Thread 的 qemu 模拟器。输入 `ulog_example` 命令，即可看到 `ulog` 例程运行结果，大致效果如下图：

```
Administrator@ARMINK E:\program\RTT_GCC\rt-thread\bsp\qemu-vexpress-a9
> qemu.bat
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
          Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
          Specify the 'raw' format explicitly to remove the restrictions.
dsound: Could not initialize DirectSoundCapture
dsound: Reason: No sound driver is available for use, or the given GUID is not a valid DirectSound device ID
\ | /
- RT - Thread Operating System
/ | \ 3.1.1 build Sep 29 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh >ulog_example
[8887] D/example: LOG_D(1): RT-Thread is an open source IoT operating system from China.
[8887] I/example: LOG_I(1): RT-Thread is an open source IoT operating system from China.
[8887] W/example: LOG_W(1): RT-Thread is an open source IoT operating system from China.
[8887] E/example: LOG_E(1): RT-Thread is an open source IoT operating system from China.
[8887] D/test: ulog_d(1): RT-Thread is an open source IoT operating system from China.
[8887] I/test: ulog_i(1): RT-Thread is an open source IoT operating system from China.
[8887] W/test: ulog_w(1): RT-Thread is an open source IoT operating system from China.
[8887] E/test: ulog_e(1): RT-Thread is an open source IoT operating system from China.
```

图 26.8: `ulog` 异步例程

大家如果细心观察可以发现，开启异步模式后，这一些在代码上离得非常近的日志的时间信息几乎是相同的。但在同步模式下，日志使用用户线程来输出，由于日志输出要花一定时间，所以每条日志的时间会有一定的间隔。这里也充分说明了异步日志的输出效率很高，几乎不占用调用者的时间。

26.5.3 日志动态过滤器

前面小节有介绍过一些日志的静态过滤功能，静态过滤有其优点比如：节省资源，但很多时候，用户需要在软件运行时动态调整日志的过滤方式，这就可以使用到 ulog 的动态过滤器功能。使用动态过滤器功能需在 `menuconfig` 中开启 `Enable runtime log filter`. 选项，该选项默认关闭。

`ulog` 支持的动态过滤方式有以下 4 种，并且都有对应的 API 函数及 Finsh/MSH 命令，下面将会逐一介绍。

26.5.3.1 按模块的级别过滤

```
int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level)
```

参数	描述
tag	日志的标签
level	设定的日志级别
返回	-
>=0	成功
-5	失败，没有足够的内存

- 命令格式：`ulog_tag_lvl <tag> <level>`

这里指的模块代表一类具有相同标签属性的日志代码。有些时候需要在运行时动态的修改某一个模块的日志输出级别。

参数 `level` 日志级别可取如下值：

级别	名称
LOG_LVL_ASSERT	断言
LOG_LVL_ERROR	错误
LOG_LVL_WARNING	警告
LOG_LVL_INFO	信息
LOG_LVL_DBG	调试
LOG_FILTER_LVL_SILENT	静默（停止输出）
LOG_FILTER_LVL_ALL	全部

函数调用与命令示例如下所示：

功能	函数调用	执行命令
关闭 wifi 模块全部日志	<code>ulog_tag_lvl_filter_set("wifi", LOG_FILTER_LVL_SILENT);</code>	<code>ulog_tag_lvl wifi 0</code>
开启 wifi 模块全部日志	<code>ulog_tag_lvl_filter_set("wifi", LOG_FILTER_LVL_ALL);</code>	<code>ulog_tag_lvl wifi 7</code>
设置 wifi 模块日志级别为警告	<code>ulog_tag_lvl_filter_set("wifi", LOG_LVL_WARNING);</code>	<code>ulog_tag_lvl wifi 4</code>

26.5.3.2 按标签全局过滤

```
void ulog_global_filter_tag_set(const char *tag)
```

参数	描述
<code>tag</code>	设定的过滤标签

- 命令格式: `ulog_tag [tag]` , `tag` 为空时, 则取消标签过滤。

该过滤方式可以对所有日志执行按标签过滤, 只有包含标签信息的日志才允许输出。

例如: 有 `wifi.driver`、`wifi.mgnt`、`audio.driver` 3 种标签的日志, 当设定过滤标签为 `wifi` 时, 只有标签为 `wifi.driver` 及 `wifi.mgnt` 的日志会输出。同理, 当设置过滤标签为 `driver` 时, 只有标签为 `wifi.driver` 及 `audio.driver` 的日志会输出。常见功能对应的函数调用与命令示例如下:

功能	函数调用	执行命令
设置过滤标签为 <code>wifi</code>	<code>ulog_global_filter_tag_set("wifi");</code>	<code>ulog_tag wifi</code>
设置过滤标签为 <code>driver</code>	<code>ulog_global_filter_tag_set("driver")</code> ;	<code>ulog_tag driver</code>
取消标签过滤	<code>ulog_global_filter_tag_set("");</code>	<code>ulog_tag</code>

26.5.3.3 按级别全局过滤

```
void ulog_global_filter_lvl_set(rt_uint32_t level)
```

参数	描述
<code>level</code>	设定的日志级别

- 命令格式: `ulog_lvl <level>` , `level` 取值参照下表:

取值	描述
0	断言
3	错误
4	警告
6	信息
7	调试

通过函数或者命令设定好全局的过滤级别以后，低于设定级别的日志都将停止输出。常见功能对应的函数调用与命令示例如下：

功能	函数调用	执行命令
关闭全部日志	<code>ulog_global_filter_lvl_set(LOG_FILTER_LVL_SILENT);</code>	<code>ulog_lvl 0</code>
开启全部日志	<code>ulog_global_filter_lvl_set(LOG_FILTER_LVL_ALL);</code>	<code>ulog_lvl 7</code>
设置日志级别为警告	<code>ulog_global_filter_lvl_set(LOG_LVL_WARNING);</code>	<code>ulog_lvl 4</code>

26.5.3.4 按关键词全局过滤

```
void ulog_global_filter_kw_set(const char *keyword)
```

参数	描述
<code>keyword</code>	设定的过滤关键词

- 命令格式：`ulog_kw [keyword]`，`keyword`为空时，则取消关键词过滤。

该过滤方式可以对所有日志执行按关键词过滤，包含关键词信息的日志才允许输出。常见功能对应的函数调用与命令示例如下：

功能	函数调用	执行命令
设置过滤关键词为 wifi	<code>ulog_global_filter_kw_set("wifi");</code>	<code>ulog_kw wifi</code>
清空过滤关键词	<code>ulog_global_filter_kw_set("");</code>	<code>ulog_kw</code>

26.5.3.5 查看过滤器信息

在设定完过滤器参数后，如果想要查看当前过滤器信息，可以输入 `ulog_filter` 命令，大致效果如下：

```
msh />ulog_filter
-----
ulog global filter:
level   : Debug
tag     : NULL
keyword : NULL
-----
ulog tag's level filter:
wifi          : Warning
audio.driver  : Error
msh />
```

!!! tip “提示” 过滤参数也支持保存在 Flash 中，也支持开机自动装载配置。如果需要该功能，请查看 **ulog_easyflash** 软件包的使用说明。（[点击查看](#)）

26.5.3.6 使用示例

依然是在 qemu BSP 中执行，首先在 menuconfig 开启动态过滤，然后保存配置并编译、运行例程，在日志输出约 **20** 次后，会执行 ulog_example.c 里对应的如下过滤代码：

```
if (count == 20)
{
    /* Set the global filer level is INFO. All of DEBUG log will stop output */
    ulog_global_filter_lvl_set(LOG_LVL_INFO);
    /* Set the test tag's level filter's level is ERROR. The DEBUG, INFO, WARNING
       log will stop output. */
    ulog_tag_lvl_filter_set("test", LOG_LVL_ERROR);
}
```

此时全局的过滤级别由于被设定到了 INFO 级别，所以无法再看到比 INFO 级别低的日志。同时，又将 **test** 标签的日志输出级别设定为 **ERROR**，此时 **test** 标签里比 **ERROR** 低的日志也都停止输出了。在每条日志里都有当前日志输出次数的计数值，对比的效果如下：

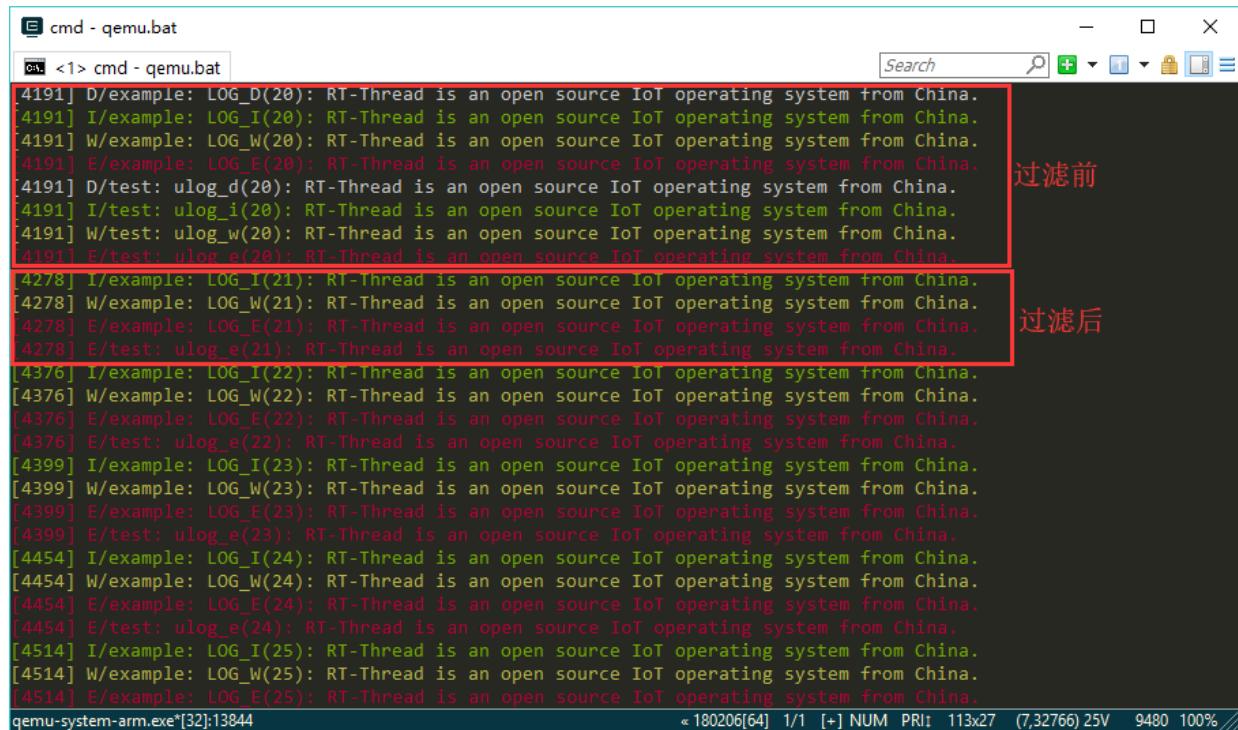
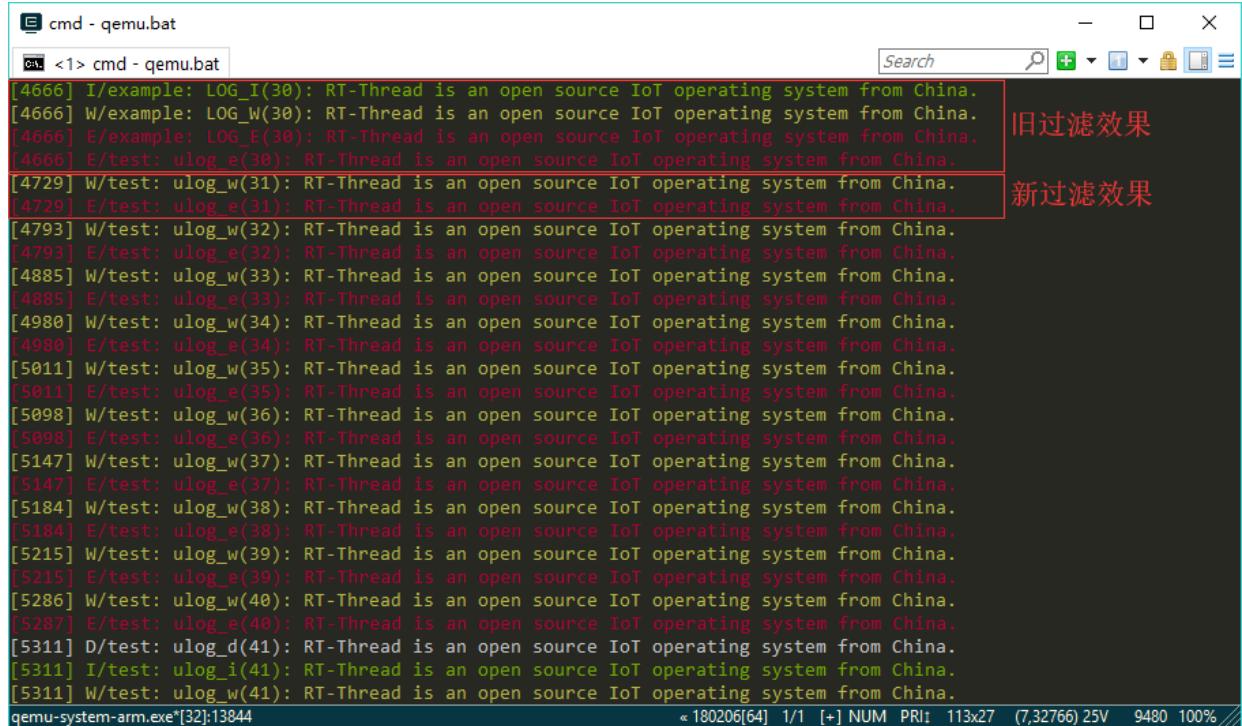


图 26.9: ulog 过滤器例程 20

在日志输出约 30 次后，会执行 `ulog_example.c` 里对应的如下过滤代码：

```
...
else if (count == 30)
{
    /* Set the example tag's level filter's level is LOG_FILTER_LVL_SILENT, the log
       enter silent mode. */
    ulog_tag_lvl_filter_set("example", LOG_FILTER_LVL_SILENT);
    /* Set the test tag's level filter's level is WARNING. The DEBUG, INFO log will
       stop output. */
    ulog_tag_lvl_filter_set("test", LOG_LVL_WARNING);
}
...
```

此时又新增了 `example` 模块的过滤器，并且是将这个模块的所有日志都停止输出，所以接下来将看不到该模块日志。同时，又将 `test` 标签的日志输出级别降低为 `WARING`，此时就只能看到 `test` 标签的 `WARING` 与 `ERROR` 级别日志。效果如下：

图 26.10: *ulog* 过滤器例程 30

在日志输出约 **40** 次后，会执行 *ulog_example.c* 里对应的如下过滤代码：

```
...
else if (count == 40)
{
    /* Set the test tag's level filter's level is LOG_FILTER_LVL_ALL. All level log
       will resume output. */
    ulog_tag_lvl_filter_set("test", LOG_FILTER_LVL_ALL);
    /* Set the global filer level is LOG_FILTER_LVL_ALL. All level log will resume
       output */
    ulog_global_filter_lvl_set(LOG_FILTER_LVL_ALL);
}
```

此时将 *test* 模块的日志输出级别调整为 *LOG_FILTER_LVL_ALL*，即不再过滤该模块任何级别的日志。同时，又将全局过滤级别设定为 *LOG_FILTER_LVL_ALL*，所以接下来 *test* 模块的全部日志将恢复输出。效果如下：

```
[5286] W/test: ulog_w(40): RT-Thread is an open source IoT operating system from China.
[5287] E/test: ulog_e(40): RT-Thread is an open source IoT operating system from China.
[5311] D/test: ulog_d(41): RT-Thread is an open source IoT operating system from China.
[5311] I/test: ulog_i(41): RT-Thread is an open source IoT operating system from China.
[5311] W/test: ulog_w(41): RT-Thread is an open source IoT operating system from China.
[5311] E/test: ulog_e(41): RT-Thread is an open source IoT operating system from China.
[5399] D/test: ulog_d(42): RT-Thread is an open source IoT operating system from China.
[5399] I/test: ulog_i(42): RT-Thread is an open source IoT operating system from China.
[5399] W/test: ulog_w(42): RT-Thread is an open source IoT operating system from China.
[5399] E/test: ulog_e(42): RT-Thread is an open source IoT operating system from China.
[5443] D/test: ulog_d(43): RT-Thread is an open source IoT operating system from China.
[5443] I/test: ulog_i(43): RT-Thread is an open source IoT operating system from China.
[5443] W/test: ulog_w(43): RT-Thread is an open source IoT operating system from China.
[5443] E/test: ulog_e(43): RT-Thread is an open source IoT operating system from China.
[5536] D/test: ulog_d(44): RT-Thread is an open source IoT operating system from China.
[5536] I/test: ulog_i(44): RT-Thread is an open source IoT operating system from China.
[5536] W/test: ulog_w(44): RT-Thread is an open source IoT operating system from China.
[5536] E/test: ulog_e(44): RT-Thread is an open source IoT operating system from China.
[5576] D/test: ulog_d(45): RT-Thread is an open source IoT operating system from China.
[5576] I/test: ulog_i(45): RT-Thread is an open source IoT operating system from China.
[5576] W/test: ulog_w(45): RT-Thread is an open source IoT operating system from China.
[5576] E/test: ulog_e(45): RT-Thread is an open source IoT operating system from China.
[5644] D/test: ulog_d(46): RT-Thread is an open source IoT operating system from China.
[5644] I/test: ulog_i(46): RT-Thread is an open source IoT operating system from China.
[5644] W/test: ulog_w(46): RT-Thread is an open source IoT operating system from China.
[5644] E/test: ulog_e(46): RT-Thread is an open source IoT operating system from China.
[5694] D/test: ulog_d(47): RT-Thread is an open source IoT operating system from China.
```

qemu-system-arm.exe*32:13844 * 180206[64] 1/1 [+/-] NUM PRI: 113x27 (8,32766) 25V 9480 100%

图 26.11: ulog 过滤器例程 40

26.5.4 系统异常时的使用

由于 ulog 的异步模式具有缓存机制，注册进来的后端内部也可能具有缓存。如果系统出现了 hardfault、断言等错误情况，但缓存中还有日志没有输出出来，这可能会导致日志丢失的问题，对于查找异常的原因会无从入手。

针对这种场景，ulog 提供了统一的日志 flush 函数：`void ulog_flush(void)`，当出现异常时，输出异常信息日志时，同时再调用该函数，即可保证缓存中剩余的日志也能够输出到后端中去。

下面以 RT-Thread 的断言及 CmBacktrace 进行举例：

26.5.4.1 断言

RT-Thread 的断言支持断言回调（hook），我们定义一个类似如下的断言 hook 函数，然后通过 `rt_assert_set_hook(rtt_user_assert_hook)`；函数将其设置到系统中即可。

```
static void rtt_user_assert_hook(const char* ex, const char* func, rt_size_t line)
{
    rt_enter_critical();

    ulog_output(LOG_LVL_ASSERT, "rtt", RT_TRUE, "(%s) has assert failed at %s:%ld.",
               ex, func, line);
    /* flush all log */
    ulog_flush();
    while(1);
}
```

26.5.4.2 CmBacktrace

CmBacktrace 是一个 ARM Cortex-M 系列 MCU 的错误诊断库，它也有对应 RT-Thread 软件包，并且最新版的软件包已经做好了针对于 uLog 的适配。里面适配代码位于 `cmb_cfg.h` :

```
...
/* print line, must config by user */
#include <rtthread.h>
#ifndef RT_USING_ULONG
#define cmb.println(...) rt_kprintf(__VA_ARGS__);rt_kprintf("\r\n")
#else
#include <ulog.h>
#define cmb.println(...) ulog_e("cmb", __VA_ARGS__);ulog_flush()
#endif /* RT_USING_ULONG */
...
```

由此可以看出，当启用了 `ulog` 以后，CmBacktrace 的每一条日志输出时都会使用错误级别，并且会同时执行 `ulog_flush`，用户无需再做任何修改。

26.5.5 syslog 模式

在 Unix 类操作系统上，syslog 广泛应用于系统日志。syslog 常见的后端有文件和网络，syslog 日志可以记录在本地文件中，也可以通过网络发送到接收 syslog 的服务器。

`ulog` 提供了 syslog 模式的支持，不仅仅前端 API 与 syslog API 完全一致，日志的格式也符合 RFC 标准。但需要注意的是，在开启 syslog 模式后，不管使用哪一种日志输出 API，整个 `ulog` 的日志输出格式都会采用 syslog 格式。

使用 `syslog` 配置需要开启 `Enable syslog format log and API.` 选项。

26.5.5.1 日志格式

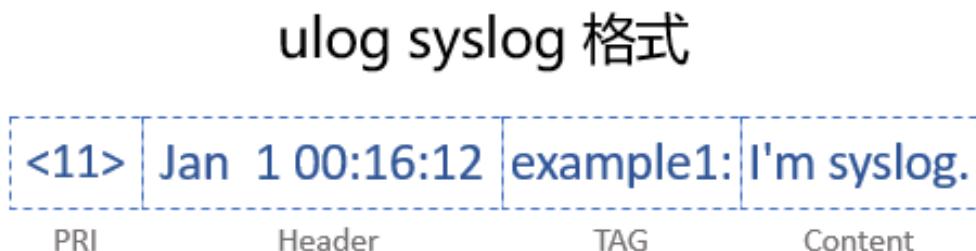


图 26.12: `ulog syslog` 格式

如上图所示，`ulog syslog` 日志格式分为下面 4 个部分：

格式	描述
PRI	PRI 部分由尖括号包含的一个数字构成，这个数字包含了程序模块（Facility）、严重性（Severity）信息，是由 Facility 乘以 8，然后加上 Severity 得来。Facility 和 Severity 由 syslog 函数的入参传入，具体数值详见 syslog.h
Header	Header 部分主要是时间戳，指示当前日志的时间；
TAG	当前日志的标签，可以通过 openlog 函数入参传入，如果不指定将会使用 rtt 作为默认标签
Content	日志的具体内容

26.5.5.2 使用方法

使用前需要在 menuconfig 中开启 syslog 选项，主要常用的 API 有：

- 打开 syslog: void openlog(const char *ident, int option, int facility)
- 输出 syslog 日志: void syslog(int priority, const char *format, ...)

!!! tip “提示” 提示：调用 openlog 是可选择的。如果不调用 openlog，则在第一次调用 syslog 时，自动调用 openlog

syslog() 函数的使用方法也非常简单，其入参格式与 printf 函数一致。在 ulog_example.c 中也有 syslog 的例程，在 qemu 中的运行效果大致如下：

```

cmd - qemu.bat
\ | /
- RT - Thread Operating System
/ | \ 3.1.1 build Sep 29 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh />ulog_example
<14>Jan 1 00:01:24 example1: syslog(1) LOG_INFO: RT-Thread is an open source IoT operating system from China.
<15>Jan 1 00:01:24 example1: syslog(1) LOG_DEBUG: RT-Thread is an open source IoT operating system from China.
<12>Jan 1 00:01:24 example1: syslog(1) LOG_WARNING: RT-Thread is an open source IoT operating system from China.
<11>Jan 1 00:01:24 example1: syslog(1) LOG_ERR: RT-Thread is an open source IoT operating system from China.
<22>Jan 1 00:01:24 example1: syslog(1) LOG_INFO | LOG_MAIL: RT-Thread is an open source IoT operating system from China.
<31>Jan 1 00:01:24 example1: syslog(1) LOG_DEBUG | LOG_DAEMON: RT-Thread is an open source IoT operating system from China.
<36>Jan 1 00:01:24 example1: syslog(1) LOG_WARNING | LOG_AUTH: RT-Thread is an open source IoT operating system from China.
<43>Jan 1 00:01:24 example1: syslog(1) LOG_ERR | LOG_SYSLOG: RT-Thread is an open source IoT operating system from China.
<14>Jan 1 00:01:25 example1: syslog(2) LOG_INFO: RT-Thread is an open source IoT operating system from China.
<15>Jan 1 00:01:25 example1: syslog(2) LOG_DEBUG: RT-Thread is an open source IoT operating system from China.
<12>Jan 1 00:01:25 example1: syslog(2) LOG_WARNING: RT-Thread is an open source IoT operating system from China.
<11>Jan 1 00:01:25 example1: syslog(2) LOG_ERR: RT-Thread is an open source IoT operating system from China.
<22>Jan 1 00:01:25 example1: syslog(2) LOG_INFO | LOG_MAIL: RT-Thread is an open source IoT operating system from China.
<31>Jan 1 00:01:25 example1: syslog(2) LOG_DEBUG | LOG_DAEMON: RT-Thread is an open source IoT operating system from China.
<36>Jan 1 00:01:25 example1: syslog(2) LOG_WARNING | LOG_AUTH: RT-Thread is an open source IoT operating system from China.
<43>Jan 1 00:01:25 example1: syslog(2) LOG_ERR | LOG_SYSLOG: RT-Thread is an open source IoT operating system from China.
<14>Jan 1 00:01:25 example1: syslog(3) LOG_INFO: RT-Thread is an open source IoT operating system from China.
* 180206[64] 1/1 [+] NUM PRI: 128x27 (7,1933) 25V 6564 100%
qemu-system-arm.exe*[32]:12496

```

图 26.13: ulog syslog 例程

26.5.6 从 rt_dbg.h 或 elog 迁移到 ulog

如果项目中以前使用的是这两类日志组件，当要使用 ulog 时，就会牵扯到如何让以前代码也支持 ulog，下面将会重点介绍迁移过程。

26.5.6.1 从 rt_dbg.h 迁移

当前 rtdbg 已完成无缝对接 ulog，开启 ulog 后，旧项目中使用 rtdbg 的代码无需做任何修改，即可使用 ulog 完成日志输出。

26.5.6.2 从 elog (EasyLogger) 迁移

如果无法确认某个源代码文件运行的目标平台上一定会使用 ulog，那么还是建议在该文件中增加下面的改动：

```
#ifdef RT_USING_ULONG
#include <ulog.h>
#else
#include <elog.h>
#endif /* RT_USING_ULONG */
```

如果明确只会使用 ulog 组件后，那么只需将头文件引用从 `elog.h` 更换为 `ulog.h`，其他任何代码都无需改动。

26.5.7 日志使用技巧

有了日志工具后，如果使用不当，也会造成日志被滥用、日志信息无法突出重点等问题。这里重点与大家分享下日志组件在使用时的一些技巧，让日志信息更加直观。主要关注点有：

26.5.7.1 合理利用标签分类

合理利用标签功能，每个模块代码在使用日志前，先明确好模块、子模块名称。这样也能让日志在最开始阶段就做好分类，为后期日志过滤也做好了准备。

26.5.7.2 合理利用日志级别

刚开始使用日志库时，大家会经常遇到警告与错误日志无法区分，信息与调试日志无法区分，导致日志级别选择不合适。一些重要日志可能看不到，不重要的日志满天飞等问题。所以，在使用前务必仔细阅读日志级别小节，针对各个级别划分，里面有明确的标准。

26.5.7.3 避免重复性冗余日志

在一些情况下会出现代码的重复调用或者循环执行，多次输出相同、相似的日志问题。这样的日志不仅会占用很大的系统资源，还会影响开发人员对于问题的定位。所以，在遇到这种情况时，建议增加对于重复性日志特殊处理，比如：让上层来输出一些业务有关的日志，底层只返回具体结果状态；同一个时间点下相同的日志，是否可以增加去重处理，在错误状态没有变化时，只输出一次等等。

26.5.7.4 开启更多的日志格式

ulog 默认的日志格式中没有开启时间戳及线程信息。这两个日志信息，在 RTOS 上挺实用。它们能帮助开发者直观的了解各个日志的运行时间点、时间差，还能清晰的看到是在哪个线程执行当前代码。所以如果条件允许，还是建议开启。

26.5.7.5 关闭不重要的日志

ulog 提供了多种维度的日志开关、过滤的功能，完全能够做到精细化控制，所以如果在调试某个功能模块时，可以适当关闭其他无关模块的日志输出，这样就可以聚焦在当前调试的模块上。

26.6 常见问题

26.6.1 Q: 日志代码已执行，但是无输出。

A: 参考日志级别小节，了解日志级别分类，并检查日志过滤参数。还有种可能是不小心将控制台后端给关闭了，重新开启 `Enable console backend` 即可。

26.6.2 Q: 开启 ulog 后，系统运行崩溃，例如：线程堆栈溢出。

A: ulog 比起以前用的 rtdbg 或者 `rt_kprintf` 打印输出函数会多占一部分线程堆栈空间，如果是开启了浮点数打印支持，由于其内部使用了 `libc` 里资源占用加大的 `vsnprintf`，所以堆栈建议多预留 250 字节。如果开启了时间戳功能，堆栈建议多预留 100 字节。

26.6.3 Q: 日志内容的末尾缺失。

A: 这是由于日志内容超出设定的日志的最大宽度。检查 `The log's max width` 选项，并增大其至合适的大小。

26.6.4 Q: 开启时间戳以后，为什么看不到毫秒级时间。

A: 这是因为 ulog 目前只支持在开启软件模拟 RTC 状态下，显示毫秒级时间戳。如需显示，只要开启 RT-Thread 软件模拟 RTC 功能即可。

26.6.5 Q: 每次 include ulog 头文件前，都要定义 LOG_TAG 及 LOG_LVL，可否简化。

A: `LOG_TAG`如果不定义，默认会使用 `NO_TAG` 标签，这样输出的日志会容易产生误解，所以标签的宏不建议省略。

`LOG_LVL`如果不定义，默认会使用调试级别，如果该模块处于开发阶段这个过程可以省略，但是模块代码如果已经稳定，建议定义该宏，并修改级别为信息级别。

26.6.6 Q: 运行出现警告提示：**Warning: There is no enough buffer for saving async log, please increase the ULOG_ASYNC_OUTPUT_BUF_SIZE option.**

A: 当遇到该提示时，说明了在异步模式下的缓冲区出现了溢出的情况，这会导致一部分日志丢失，增大 `ULOG_ASYNC_OUTPUT_BUF_SIZE` 选项可以解决该问题。

26.6.7 Q: 编译时提示: The idle thread stack size must more than 384 when using async output by idle (ULOG_ASYNC_OUTPUT_BY_IDLE)。

A: 在使用 idle 线程作为输出线程时, idle 线程的堆栈大小需要提高, 这也取决于具体的后端设备, 例如: 控制台后端时, idle 线程至少得 384 字节。

第 27 章

utest 测试框架

27.1 utest 简介

utest (unit test) 是 RT-Thread 开发的单元测试框架。设计 utest 的初衷是方便 RT-Thread 开发者使用统一的框架接口编写测试程序，实现单元测试、覆盖测试以及集成测试的目的。

27.1.1 测试用例定义

测试用例 (testcase, 简称 tc) 是为实现特定测试目标而执行的单个测试，是包括测试输入、执行条件、测试过程和预期结果的规范，是一个有明确的结束条件和明确的测试结果的有限循环。

utest (unit test) 测试框架定义用户编写的测试程序为测试用例，一个测试用例仅包含一个 *testcase* 函数（类似 main 函数），可包含多个测试单元函数。

具体地通过 utest 测试框架提供的 API 完成的针对某一功能的测试代码就是一个测试用例。

27.1.2 测试单元定义

测试单元 (test unit) 是被测功能细分后的测试点，每个测试点可以任务是被测功能的最小可测单位。当然，不同的分类方式会细分出不同的测试单元。

27.1.3 utest 应用框图

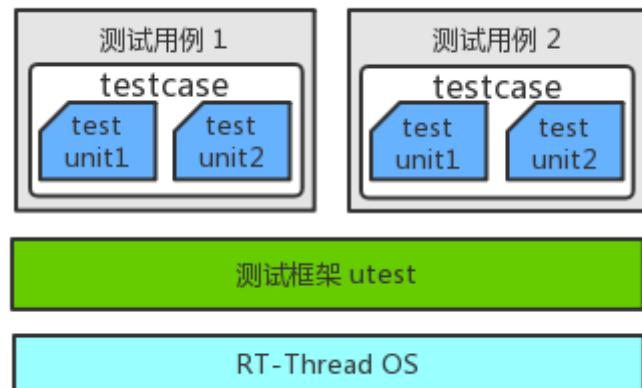


图 27.1: utest 应用框图

如上图所示，测试用例基于测试框架 utest 测试框架提供的服务接口进行程序设计，支持将多个测试用例编译到一起进行测试。另外从图中可以看到，一个测试用例对应唯一的 `testcase` 函数，在 `testcase` 中包含多个测试单元（`test unit`）。

27.2 utest API

为了能够实现格式统一的测试用例代码，测试框架 utest 为测试用例编写提供了一套通用的 API 接口。

27.2.1 assert 宏

注意：

这里的 `assert` 仅记录通过和失败的数量，不会产生断言并终止程序运行。其功能不等同于 `RT_ASSERT`。

assert 宏	说明
<code>uassert_true(value)</code>	<code>value</code> 为 <code>true</code> 则测试通过，否则测试失败
<code>uassert_false(value)</code>	<code>value</code> 为 <code>false</code> 则测试通过，否则测试失败
<code>uassert_null(value)</code>	<code>value</code> 为 <code>null</code> 则测试通过，否则测试失败
<code>uassert_not_null(value)</code>	<code>value</code> 为非 <code>null</code> 值则测试通过，否则测试失败
<code>uassert_int_equal(a, b)</code>	<code>a</code> 和 <code>b</code> 值相等则测试通过，否则测试失败
<code>uassert_int_not_equal(a, b)</code>	<code>a</code> 和 <code>b</code> 值不相等则测试通过，否则测试失败
<code>uassert_str_equal(a, b)</code>	字符串 <code>a</code> 和字符串 <code>b</code> 相同则测试通过，否则测试失败
<code>uassert_str_not_equal(a, b)</code>	字符串 <code>a</code> 和字符串 <code>b</code> 不相同则测试通过，否则测试失败

assert 宏	说明
uassert_in_range(value, min, max)	value 在 min 和 max 的范围内则测试通过，否则测试失败
uassert_not_in_range(value, min, max)	value 不在 min 和 max 的范围内则测试通过，否则测试失败

27.2.2 测试单元运行宏

```
UTEST_UNIT_RUN(test_unit_func)
```

测试用例中，使用 UTEST_UNIT_RUN 宏执行指定的测试单元函数 test_unit_func。测试单元（test unit）必须使用 UTEST_UNIT_RUN 宏执行。

27.2.3 测试用例导出宏

```
UTEST_TC_EXPORT(testcase, name, init, cleanup, timeout)
```

参数	描述
testcase	测试用例主承载函数（规定使用名为 static void testcase(void) 的函数）
name	测试用例名称（唯一性）。规定使用测试用例相对 testcases 目录的相对路径以 . 进行连接的命名格式
init	测试用例启动前的初始化函数
cleanup	测试用例结束后的清理函数
timeout	测试用例预计需要的测试时间（单位是秒）

测试用例命名要求：

测试用例需要按照规定的格式命名。规定使用当前测试用例相对 testcases 目录的相对路径以 . 进行连接的命名格式，名字中包含当前测试用例文件的文件名（除去后缀名的文件名）。

测试用例命名示例：

假设在测试用例 testcases 目录下，有 testcases\components\filesystem\dfs\dfs_api_tc.c 测试用例文件，那么该 dfs_api_tc.c 中的测试用例的名称命名为 components.filesystem.dfs.dfs_api_tc。

27.2.4 测试用例 LOG 输出接口

utest 测试框架依赖 ulog 日志模块进行日志输出，并且 utest 测试框架中已经日志输出级别。因此只要在测试用例里加入 `##include "utest.h"` 即可使用 ulog 日志模块的所有级别接口（LOG_D/LOG_I/LOG_E）。

另外，`utest` 测试框架增加了额外的日志控制接口，如下：

```
#define UTEST_LOG_ALL      (1u)
#define UTEST_LOG_ASSERT   (2u)

void utest_log_lv_set(rt_uint8_t lv);
```

用户可以在测试用例中使用 `utest_log_lv_set` 接口控制日志输出级别。`UTEST_LOG_ALL` 配置输出所有日志，`UTEST_LOG_ASSERT` 配置仅输出 `uassert` 失败后的日志。

27.3 配置使能

使用 `utest` 测试框架需要在 ENV 工具中使用 `menuconfig` 进行如下配置：

```
RT-Thread Kernel --->
  Kernel Device Object --->
    (256) the buffer size for console log printf /* utest 日志需要的最小 buffer */
          */
RT-Thread Components --->
  Utilities --->
    -*- Enable utest (RT-Thread test framework) /* 使能 utest 测试框架 */
    (4096) The utest thread stack size           /* 设置 utest 线程堆栈 (-thread
          模式需要) */
    (20)   The utest thread priority             /* 设置 utest 线程优先级 (-
          thread 模式需要) */
```

27.4 应用范式

前面介绍了 `utest` 测试框架和相关 API，这里介绍基本的测试用例代码结构。

测试用例文件必须的代码块如下所示：

```
/*
 * Copyright (c) 2006-2019, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date       Author    Notes
 * 2019-01-16  MurphyZhao the first version
 */

#include <rtthread.h>
#include "utest.h"

static void test_xxx(void)
{
```

```
    uassert_true(1);
}

static rt_err_t utest_tc_init(void)
{
    return RT_EOK;
}

static rt_err_t utest_tc_cleanup(void)
{
    return RT_EOK;
}

static void testcase(void)
{
    UTEST_UNIT_RUN(test_xxx);
}

UTEST_TC_EXPORT(testcase, "components.utilities.utest.sample.sample_tc",
    utest_tc_init, utest_tc_cleanup, 10);
```

一个基本的测试用例必须包含以下内容：

- 文件注释头（Copyright）

测试用例文件必须包含文件注释头，包含 Copyright、时间、作者、描述信息。

- utest_tc_init(void)

测试运行前的初始化函数，一般用来初始化测试需要的环境。

- utest_tc_cleanup(void)

测试结束后的清理函数，用来清理测试过程中申请的资源（如内存，线程，信号量等）。

- testcase(void)

测试主体函数，一个测试用例实现仅能包含一个 testcase 函数（类似 main 函数）。通常该函数里只用来运行测试单元执行函数 UTEST_UNIT_RUN。

一个 testcase 中可以包含多个测试单元，每个测试单元由 UTEST_UNIT_RUN 执行。

- UTEST_UNIT_RUN

测试单元执行函数。

- test_xxx(void)

每个功能单元的测试实现。用户根据需求确定函数名和函数实现。

- uassert_true

用于判断测试结果的断言宏（该断言宏并不会终止程序运行）。测试用例必须使用 uassert_xxx 宏来判断测试结果，否则测试框架不知道测试是否通过。

所有的 uassert_xxx 宏都通过后，整个测试用例才算测试通过。

- UTEST_TC_EXPORT

将测试用例 testcase 函数导出到测试框架。

27.5 测试用例运行要求

测试框架 `utest` 将所有的测试用例导出到了 `UtestTcTab` 代码段，在 IAR 和 MDK 编译器中不需要在链接脚本中定义 `UtestTcTab` 段，但是在 GCC 编译时，需要在链接脚本中显式地设置 `UtestTcTab` 段。

因此，测试用例要在 GCC 下能够编译运行，必须要先在 GCC 的链接脚本中定义 `UtestTcTab` 代码段。

在 GCC 链接脚本的 `.text` 中，增加 `UtestTcTab` 段的定义，格式如下所示：

```
/* section information for utest */
. = ALIGN(4);
__rt_utest_tc_tab_start = .;
KEEP(*(UtestTcTab))
__rt_utest_tc_tab_end = .;
```

27.6 运行测试用例

测试框架提供了以下命令，便于用户在 RT-Thread MSH 命令行中运行测试用例，命令如下：

`utest_list` 命令

列出当前系统支持的测试用例，包括测试用例的名称和测试需要的时间。该命令无参数。

`utest_run` 命令

测试用例执行命令，该命令格式如下：

```
utest_run [-thread or -help] [ testcase name ] [ loop num ]
```

utest_run 命令参数	描述
<code>-thread</code>	使用线程模式运行测试框架
<code>-help</code>	打印帮助信息
<code>testcase name</code>	指定测试用例名称。支持使用通配符*，支持指定测试用例名称前部分字节
<code>loop num</code>	指定测试用例循环测试次数

测试命令使用示例：

```
msh />utest_list
[14875] I/utest: Commands list :
[14879] I/utest: [ testcase name ]:components.filesystem.dfs.dfs_api_tc; [run timeout
]:30
[14889] I/utest: [ testcase name ]:components.filesystem.posix posix_api_tc; [run
timeout]:30
[14899] I/utest: [ testcase name ]:packages.iot.netutils.iperf iperf_tc; [run timeout
]:30
msh />
```

```
msh />utest_run components.filesystem.dfs.dfs_api_tc
[83706] I/utest: [=====] [ utest ] started
[83712] I/utest: [-----] [ testcase ] (components.filesystem.dfs.dfs_api_tc)
    started
[83721] I/testcase: in testcase func...
[84615] D/utest: [    OK     ] [ unit      ] (test_mkfs:26) is passed
[84624] D/testcase: dfs mount rst: 0
[84628] D/utest: [    OK     ] [ unit      ] (test_dfs_mount:35) is passed
[84639] D/utest: [    OK     ] [ unit      ] (test_dfs_open:40) is passed
[84762] D/utest: [    OK     ] [ unit      ] (test_dfs_write:74) is passed
[84770] D/utest: [    OK     ] [ unit      ] (test_dfs_read:113) is passed
[85116] D/utest: [    OK     ] [ unit      ] (test_dfs_close:118) is passed
[85123] I/utest: [  PASSED   ] [ result    ] testcase (components.filesystem.dfs.
    dfs_api_tc)
[85133] I/utest: [-----] [ testcase ] (components.filesystem.dfs.dfs_api_tc)
    finished
[85143] I/utest: [=====] [ utest ] finished
msh />
```

27.6.1 测试结果分析

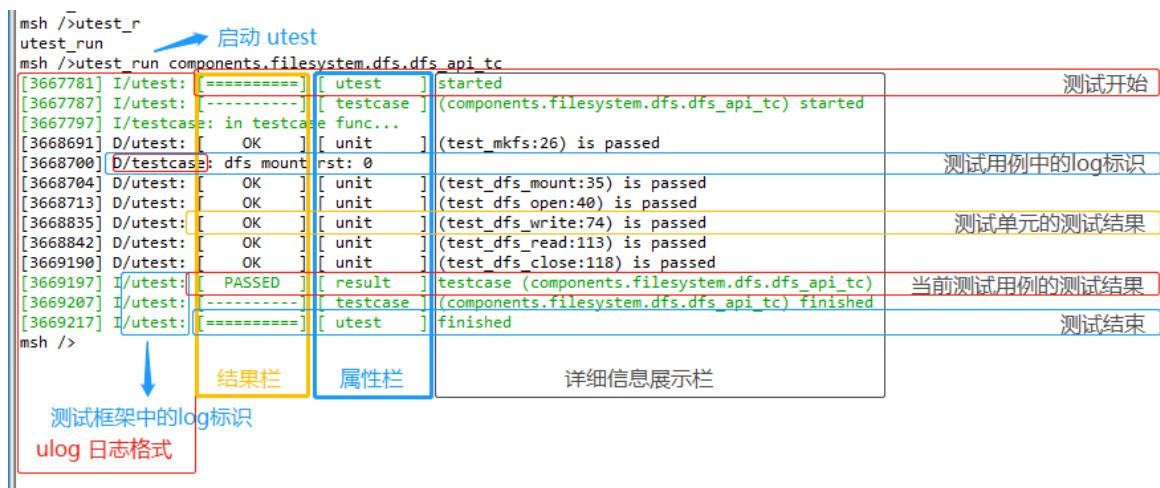


图 27.2: utest 日志展示

如上图所示，测试用例运行的日志从左到右被分成了四列，分别是 log 日志头信息、结果栏、属性栏、详细信息展示栏。日志中使用 `result` 属性标识该测试用例测试结果（PASSED or FAILED）。

27.7 测试用例运行流程

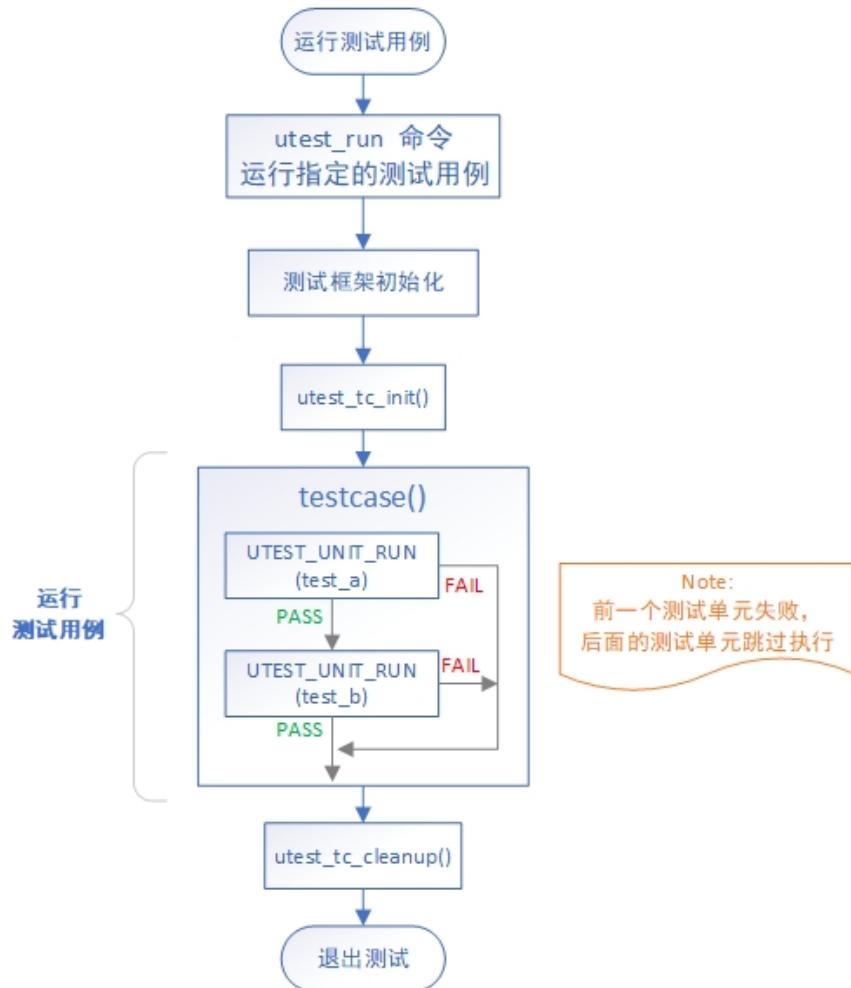


图 27.3: 测试用例运行流程

从上面的流程图中可以得到以下内容：

- utest 测试框架是顺序执行 `testcase` 函数中的所有测试单元
- 上一个 `UTEST_UNIT_RUN` 宏块出现了 assert，后面的所有 `UTEST_UNIT_RUN` 会跳过执行

27.8 注意事项

!!! note “注意事项” - 使用 GCC 编译前，确定链接脚本是否增加了 `UtestTcTab` 段 - 编译前确保 RT-Thread Kernel -> Kernel Device Object -> (256)`the buffer size for console log print` 至少为 256 字节 - 测试用例中创建的资源（线程、信号量、定时器、内存等）需要在测试结束前释放 - 一个测试用例实现仅能使用 `UTEST_TC_EXPORT` 导出一个测试主体函数（`testcase` 函数） - 为编写的测试用例程序编写 `README.md` 文档，指导用户配置测试环境

第 28 章

动态模块

在传统桌面操作系统中，用户空间和内核空间是分开的，应用程序运行在用户空间，内核以及内核模块则运行于内核空间，其中内核模块可以动态加载与删除以扩展内核功能。`dlmodule` 则是 RT-Thread 下，在内核空间对外提供的动态模块加载机制的软件组件。在 RT-Thread v3.1.0 以前的版本中，这也称之为应用模块（Application Module），在 RT-Thread v3.1.0 及之后，则回归传统，以动态模块命名。

`dlmodule` 组件更多的是一个 ELF 格式加载器，把单独编译的一个 elf 文件的代码段，数据段加载到内存中，并对其中的符号进行解析，绑定到内核导出的 API 地址上。动态模块 elf 文件主要放置于 RT-Thread 下的文件系统上。

28.1 功能简介

动态模块为 RT-Thread 提供了动态加载程序模块的机制，因为也独立于内核编译，所以使用方式比较灵活。从实现上讲，这是一种将内核和动态模块分开的机制，通过这种机制，内核和动态模块可以分开编译，并在运行时通过内核中的模块加载器将编译好的动态模块加载到内核中运行。

在 RT-Thread 的动态模块中，目前支持两种格式：

- .mo 则是编译出来时以 .mo 做为后缀名的可执行动态模块；它可以被加载，并且系统中会自动创建一个主线程执行这个动态模块中的 `main` 函数；同时这个 `main(int argc, char**argv)` 函数也可以接受命令行上的参数。
- .so 则是编译出来时以 .so 做为后缀名的动态库；它可以被加载，并驻留在内存中，并提供一些函数集由其他程序（内核里的代码或动态模块）来使用。

当前 RT-Thread 支持动态模块的架构主要包括 ARM 类架构和 x86 架构，未来会扩展到 MIPS，以及 RISC-V 等架构上。RT-Thread 内核固件部分可使用多种编译器工具链，如 GCC, ARMCC、IAR 等工具链；但动态模块部分编译当前只支持 GNU GCC 工具链编译。因此编译 RT-Thread 模块需下载 GCC 工具，例如 CodeSourcery 的 arm-none-eabi 工具链。一般的，最好内核和动态模块使用一样的工具链进行编译（这样不会在 `libc` 上产生不一致的行为）。另外，动态模块一般只能加载到 RAM 中使用，并进行符号解析绑定到内核导出的 API 地址上，而不能基于 Flash 直接以 XIP 方式运行（因为 Flash 上也不能够再行修改其中的代码段）。

28.2 使用动态模块

当要在系统中测试使用动态模块，需要编译一份支持动态模块的固件，以及需要运行的动态模块。下面将固件和动态模块的编译方式分为两部分进行介绍。

28.2.1 编译固件

当要使用动态模块时，需要在固件的配置中打开对应的选项，使用 menuconfig 打开如下配置：

```
RT-Thread Components --->
    POSIX layer and C standard library --->
        [*] Enable dynamic module with dlopen/dlsym/dlclose feature
```

也要打开文件系统的配置选项：

```
RT-Thread Components --->
    Device virtual file system --->
        [*] Using device virtual file system
```

bsp 对应的 rtconfig.py 中设置动态模块编译时需要用到的配置参数：

```
M_CFLAGS = CFLAGS + ' -mlong-calls -fPIC '
M_CXXFLAGS = CXXFLAGS + ' -mlong-calls -fPIC'
M_LFLAGS = DEVICE + CXXFLAGS + ' -Wl,--gc-sections,-z,max-page-size=0x4' +\
           ' -shared -fPIC -nostartfiles -nostdlib -static-
           libgcc'
M_POST_ACTION = STRIP + ' -R .hash $TARGET\n' + SIZE + ' $TARGET \n'
M_BIN_PATH = r'E:\qemu-dev310\fatdisk\root'
```

相关的解释如下：

- **M_CFLAGS** - 动态模块编译时用到的 C 代码编译参数，一般此处以 PIC 方式进行编译（即代码地址支持浮动方式执行）；
- **M_CXXFLAGS** - 动态模块编译时用到的 C++ 代码编译参数，参数和上面的 **M_CFLAGS** 类似；
- **M_LFLAGS** - 动态模块进行链接时的参数。同样是 PIC 方式，并且是按照共享库方式链接（部分链接）；
- **M_POST_ACTION** - 动态模块编译完成后要进行的动作，这里会对 elf 文件进行 strip 下，以减少 elf 文件的大小；
- **M_BIN_PATH** - 当动态模块编译成功时，对应的动态模块文件是否需要复制到统一的地方；

基本上来说，ARM9、Cortex-A、Cortex-M 系列的这些编译配置参数是一样的。

内核固件也会通过 RTM(function) 的方式导出一些函数 API 给动态模块使用，这些导出符号可以在 msh 下通过命令：

```
list_symbols
```

列出固件中所有导出的符号信息。**dmodule** 加载器也是把动态模块中需要解析的符号按照这里导出的符号表来进行解析，完成最终的绑定动作。

这段符号表会放在一个专门的，名字是 RTMSymTab 的 section 中，所以对应的固件链接脚本也需要保留这块区域，而不会被链接器优化移除掉。可以在链接脚本中添加对应的信息：

```
/* section information for modules */
. = ALIGN(4);
__rtmsymtab_start = .;
KEEP(*(RTMSymTab))
__rtmsymtab_end = .;
```

然后在 BSP 工程目录下执行 scons 正确无误地生成固件后，在 BSP 工程目录下执行一下命令：

```
scons --target=ua -s
```

来生成编译动态模块时需要包括的内核头文件搜索路径及全局宏定义。

28.2.2 编译动态模块

在 github 上有一份独立仓库：rtthread-apps，这份仓库中放置了一些和动态模块，动态库相关的示例。

其目录结构如下：

目录名	说明
cxx	演示了如何在动态模块中使用 C++ 进行编程
hello	最简单的 hello world 示例
lib	动态库的示例
md5	为一个文件产生 md5 码
tools	动态模块编译时需要使用到的 Python/SConscript 脚本
ymodem	通过串口以 YModem 协议下载一个文件到文件系统上

可以把这份 git clone 到本地，然后在命令行下以 scons 工具进行编译，如果是 Windows 平台，推荐使用 RT-Thread/ENV 工具。

进入控制台命令行后，进入到这个 rtthread-apps repo 所在的目录（同样的，请保证这个目录所在全路径不包含空格，中文字符等字符），并设置好两个变量：

- RTT_ROOT - 指向到 RT-Thread 代码的根目录；
- BSP_ROOT - 指向到 BSP 的工程目录；

Windows 下可以使用（假设使用的 BSP 是 qemu-vexpress-a9）：

```
set RTT_ROOT=d:\your_rtthread
set BSP_ROOT=d:\your_rtthread\bsp\qemu-vexpress-a9
```

来设置对应的环境变量。然后使用如下命令来编译动态模块，例如 hello 的例子：

```
scons --app=hello
```

编译成功后，它会在 rtthread-apps/hello 目录下生成 hello.mo 文件。

也可以编译动态库，例如 lib 的例子：

```
scons --lib=lib
```

编译成功后，它会在 rtthread-apps/lib 目录下生成 lib.so 文件。

我们可以把这些 mo、so 文件放到 RT-Thread 文件系统下。在 msh 下，可以简单的以 hello 命令方式执行 hello.mo 动态模块：

```
msh />ls
Directory /:
hello.mo          1368
lib.so            1376
msh />hello
msh />Hello, world
```

调用 hello 后，会执行 hello.mo 里的 main 函数，执行完毕后退出对应的动态模块。其中 hello/main.c 的代码如下：

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, world\n");

    return 0;
}
```

28.3 RT-Thread 动态模块 API

除了可以通过 msh 直接加载并执行动态模块外，也可以在主程序中使用 RT-Thread 提供的动态模块 API 来加载或卸载动态模块。

28.3.1 加载动态模块

```
struct rt_dlmodule *dlmodule_load(const char* pgname);
```

参数	描述
pgname	动态模块的路径
返回	—
动态模块指针	成功加载
RT_NULL	失败

这个函数从文件系统中加载动态模块到内存中，若正确加载返回该模块的指针。这个函数并不会创建一个线程去执行这个动态模块，仅仅把模块加载到内存中，并解析其中的符号地址。

28.3.2 执行动态模块

```
struct rt_dlmodule *dlmodule_exec(const char* pgname, const char* cmd, int cmd_size)
;
```

参数	描述
pgname	动态模块的路径
cmd	包括动态模块命令自身的命令行字符串
cmd_size	命令行字符串大小
返回	—
动态模块指针	成功运行
RT_NULL	失败

这个函数根据 `pgname` 路径加载动态模块，并启动一个线程来执行这个动态模块的 `main` 函数，同时 `cmd` 会作为命令行参数传递给动态模块的 `main` 函数入口。

28.3.3 退出动态模块

```
void dlmodule_exit(int ret_code);
```

参数	描述
ret_code	模块的返回参数

这个函数由模块运行时调用，它可以设置模块退出的返回值 `ret_code`，然后从模块退出。

28.3.4 查找动态模块

```
struct rt_dlmodule *dlmodule_find(const char *name);
```

参数	描述
name	模块名称
返回	—
动态模块指针	成功

参数	描述
RT_NULL	失败

这个函数以 `name` 查找系统中是否已经有加载的动态模块。

28.3.5 返回动态模块

```
struct rt_dlmodule *dlmodule_self(void);
```

参数	描述
返回	—
动态模块指针	成功
RT_NULL	失败

这个函数返回调用上下文环境下动态模块的指针。

28.3.6 查找符号

```
rt_uint32_t dlmodule_symbol_find(const char *sym_str);
```

参数	描述
sym_str	符号名称
返回	—
符号地址	成功
0	失败

这个函数根据符号名称返回符号地址。

28.4 标准 POSIX 动态库 libdl API

在 RT-Thread `dlmodule` 中也支持 POSIX 标准的 `libdl` API，类似于把一个动态库加载到内存中（并解析其中的一些符号信息），由这份动态库提供对应的函数操作集。`libdl` API 需要包含的头文件：

```
##include <dlfcn.h>
```

28.4.1 打开动态库

```
void * dlopen (const char * pathname, int mode);
```

参数	描述
pathname	动态库路径名称
mode	打开动态库时的模式，在 RT-Thread 中并未使用
返回	—
动态库的句柄 (<code>struct dlmodule</code> 结构体指针)	成功
NULL	失败

这个函数类似 `dlmodule_load` 的功能，会从文件系统上加载动态库，并返回动态库的句柄指针。

28.4.2 查找符号

```
void* dlsym(void *handle, const char *symbol);
```

参数	描述
handle	动态库句柄， <code>dlopen</code> 的返回值
symbol	要返回的符号地址
返回	—
符号地址	成功
NULL	失败

这个函数在动态库 `handle` 中查找是否存在 `symbol` 的符号，如果存在返回它的地址。

28.4.3 关闭动态库

```
int dlclose (void *handle);
```

参数	描述
handle	动态库句柄
返回	—
0	成功
负数	失败

这个函数会关闭 `handle` 指向的动态库，从内存中卸载掉。需要注意的是，当动态库关闭后，原来通过 `dlsym` 返回的符号地址将不再可用。如果依然尝试去访问，可能会引起 `fault` 错误。

28.5 常见问题

28.5.1 Env 工具的相关问题请参考《Env 用户手册》。

28.5.2 Q: 根据文档不能成功运行动态模块。

A: 请更新 RT-Thread 源代码到 3.1.0 及以上版本。

28.5.3 Q: 使用 scons 命令编译工程, 提示 undefined reference to __rtmsymtab_start。

A: 请参考 qemu-vexpress-a9 BSP 的 GCC 链接脚本文件 `link.lds`，在工程的 GCC 链接脚本的 TEXT 段增加以下内容。

```
/* section information for modules */
. = ALIGN(4);
__rtmsymtab_start = .;
KEEP(*(RTMSymTab))
__rtmsymtab_end = .;
```

第 29 章

POSIX 接口

29.1 Pthreads 简介

POSIX Threads 简称 Pthreads，POSIX 是“Portable Operating System Interface”（可移植操作系统接口）的缩写，POSIX 是 IEEE Computer Society 为了提高不同操作系统的兼容性和应用程序的可移植性而制定的一套标准。Pthreads 是线程的 POSIX 标准，被定义在 POSIX.1c, Threads extensions (IEEE Std1003.1c-1995) 标准里，该标准定义了一套 C 程序语言的类型、函数和常量。定义在 `pthread.h` 头文件和一个线程库里，大约有 100 个 API，所有 API 都带有“`pthread_`”前缀，可以分为 4 大类：

- 线程管理 (Thread management): 包括线程创建 (creating)、分离 (detaching)、连接 (joining) 及设置和查询线程属性的函数等。
- 互斥锁 (Mutex): “mutual exclusion”的缩写，用了限制线程对共享数据的访问，保护共享数据的完整性。包括创建、销毁、锁定和解锁互斥锁及一些用于设置或修改互斥量属性等函数。
- 条件变量 (Condition variable): 用于共享一个互斥量的线程间的通信。包括条件变量的创建、销毁、等待和发送信号 (signal) 等函数。
- 读写锁 (read/write lock) 和屏障 (barrier): 包括读写锁和屏障的创建、销毁、等待及相关属性设置等函数。
- POSIX 信号量 (semaphore) 和 Pthreads 一起使用，但不是 Pthreads 标准定义的一部分，被定义在 POSIX.1b, Real-time extensions (IEEE Std1003.1b-1993) 标准里。因此信号量相关函数的前缀是“`sem_`”而不是“`pthread_`”。
- 消息队列 (Message queue) 和信号量一样，和 Pthreads 一起使用，也不是 Pthreads 标准定义的一部分，被定义在 IEEE Std 1003.1-2001 标准里。消息队列相关函数的前缀是“`mq_`”。

函数前缀	函数组
<code>pthread_</code>	线程本身和各种相关函数
<code>pthread_attr_</code>	线程属性对象
<code>Pthread_mutex_</code>	互斥锁

函数前缀	函数组
<code>pthread_mutexattr_</code>	互斥锁属性对象
<code>pthread_cond_</code>	条件变量
<code>pthread_condattr_</code>	条件变量属性对象
<code>pthread_rwlock_</code>	读写锁
<code>pthread_rwlockattr_</code>	读写锁属性对象
<code>pthread_spin_</code>	自旋锁
<code>pthread_barrier_</code>	屏障
<code>pthread_barrierattr_</code>	屏障属性对象
<code>sem_</code>	信号量
<code>mq_</code>	消息队列

绝大部分 Pthreads 的函数执行成功则返回 0 值，不成功则返回一个包含在 `errno.h` 头文件中的错误代码。很多操作系统都支持 Pthreads，比如 Linux、MacOSX、Android 和 Solaris，因此使用 Pthreads 的函数编写的应用程序有很好的可移植性，可以在很多支持 Pthreads 的平台上直接编译运行。

29.1.1 在 RT-Thread 中使用 POSIX

在 RT-Thread 中使用 POSIX API 接口包括几个部分：libc（例如 newlib），filesystem，pthread 等。需要在 `rtconfig.h` 中打开相关的选项：

```
#define RT_USING_LIBC
#define RT_USING_DFS
#define RT_USING_DFS_DEVFS
#define RT_USING_PTHREADS
```

RT-Thread 实现了 Pthreads 的大部分函数和常量，按照 POSIX 标准定义在 `pthread.h`、`mqueue.h`、`semaphore.h` 和 `sched.h` 头文件里。Pthreads 是 libc 的一个子库，RT-Thread 中的 Pthreads 是基于 RT-Thread 内核函数的封装，使其符合 POSIX 标准。后续章节会详细介绍 RT-Thread 中实现的 Pthreads 函数及相关功能。

29.2 线程

29.2.1 线程句柄

```
typedef rt_thread_t pthread_t;
```

`pthread_t` 是 `rt_thread_t` 类型的重定义，定义在 `pthread.h` 头文件里。`rt_thread_t` 是 RT-Thread 的线程句柄（或线程标识符），是指向线程控制块的指针。在创建线程前需要先定义一个 `pthread_t` 类型的变量。每个线程都对应了自己的线程控制块，线程控制块是操作系统用于控制线程的一个数据结构，它存放

了线程的一些信息，例如优先级，线程名称和线程堆栈地址等。线程控制块及线程具体信息在 RT-Thread 编程手册的线程调度与管理一章有详细的介绍。

29.2.2 创建线程

```
int pthread_create (pthread_t *tid,
                    const pthread_attr_t *attr,
                    void *(*start) (void *), void *arg);
```

参数	描述
tid	指向线程句柄(线程标识符)的指针，不能为 NULL
attr	指向线程属性的指针，如果使用 NULL，则使用默认的线程属性
start	线程入口函数地址
arg	传递给线程入口函数的参数
返回	—
0	成功
EINVAL	参数无效
ENOMEM	动态分配内存失败

此函数创建一个 pthread 线程。此函数会动态分配 POSIX 线程数据块和 RT-Thread 线程控制块，并把线程控制块的起始地址（线程 ID）保存在参数 tid 指向的内存里，此线程标识符可用于在其他线程中操作此线程；并把 attr 指向的线程属性、start 指向的线程入口函数及入口函数参数 arg 保存在线程数据块和线程控制块里。如果线程创建成功，线程立刻进入就绪态，参与系统的调度，如果线程创建失败，则会释放之前线程占有的资源。

关于线程属性及相关函数会在线程高级编程一章里有详细介绍，一般情况下采用默认属性就可以。

!!! note “注意事项” 创建出 pthread 线程后，如果线程需要重复创建使用，需要设置 pthread 线程为 detach 模式，或者使用 pthread_join 等待创建后的 pthread 线程结束。

29.2.2.1 创建线程示例代码

以下程序会初始化 2 个线程，它们拥有共同的入口函数，但是它们的入口参数不相同。其他的，它们具备相同的优先级，并以时间片进行轮转调度。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;
```

```
/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程入口函数 */
static void* thread_entry(void* parameter)
{
    int count = 0;
    int no = (int) parameter; /* 获得线程的入口参数 */

    while (1)
    {
        /* 打印输出线程计数值 */
        printf("thread%d count: %d\n", no, count ++);

        sleep(2); /* 休眠 2 秒 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;

    /* 创建线程 1, 属性为默认值, 入口函数是 thread_entry, 入口函数参数是 1 */
    result = pthread_create(&tid1,NULL,thread_entry,(void*)1);
    check_result("thread1 created", result);

    /* 创建线程 2, 属性为默认值, 入口函数是 thread_entry, 入口函数参数是 2 */
    result = pthread_create(&tid2,NULL,thread_entry,(void*)2);
    check_result("thread2 created", result);

    return 0;
}
```

29.2.3 脱离线程

```
int pthread_detach (pthread_t thread);
```

参数	描述
thread	线程句柄（线程标识符）
返回	——
0	成功

调用此函数，如果 `pthread` 线程没有结束，则将 `thread` 线程属性的分离状态设置为 `detached`；当 `thread` 线程已经结束时，系统将回收 `pthread` 线程占用的资源。

使用方法：子线程调用 `pthread_detach(pthread_self())` (`pthread_self()` 返回当前调用线程的线程句柄)，或者其他线程调用 `pthread_detach(thread_id)`。关于线程属性的分离状态会在后面详细介绍。

!!! note “注意事项”一旦线程属性的分离状态设置为 `detached`，该线程不能被 `pthread_join()` 函数等待或者重新被设置为 `detached`。

29.2.3.1 脱离线程示例代码

以下程序会初始化 2 个线程，它们拥有相同的优先级，并按照时间片轮转调度。2 个线程都会被设置为脱离状态，2 个线程循环打印 3 次信息后自动退出，退出后系统将会自动回收其资源。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    int i;

    printf("i'm thread1 and i will detach myself!\n");
    pthread_detach(pthread_self());           /* 线程 1 脱离自己 */
}
```

```
for (i = 0;i < 3;i++) /* 循环打印 3 次信息 */
{
    printf("thread1 run count: %d\n",i);
    sleep(2); /* 休眠 2 秒 */
}

printf("thread1 exited!\n");
return NULL;
}

/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int i;

    for (i = 0;i < 3;i++) /* 循环打印 3 次信息 */
    {
        printf("thread2 run count: %d\n",i);
        sleep(2); /* 休眠 2 秒 */
    }

    printf("thread2 exited!\n");
    return NULL;
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;

    /* 创建线程 1, 属性为默认值, 分离状态为默认值 joinable,
     * 入口函数是 thread1_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid1,NULL,thread1_entry,NULL);
    check_result("thread1 created",result);

    /* 创建线程 2, 属性为默认值, 分离状态为默认值 joinable,
     * 入口函数是 thread2_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid2,NULL,thread2_entry,NULL);
    check_result("thread2 created",result);

    pthread_detach(tid2); /* 脱离线程 2 */

    return 0;
}
```

29.2.4 等待线程结束

```
int pthread_join (pthread_t thread, void**value_ptr);
```

参数	描述
thread	线程句柄（线程标识符）
value_ptr	用户定义的指针，用来存储被等待线程的返回值地址，可由函数 <code>pthread_join()</code> 获取
返回	——
0	成功
EDEADLK	线程 join 自己
EINVAL	join 一个分离状态为 <code>detached</code> 的线程
ESRCH	找不到 <code>thread</code> 线程

此函数会使调用该函数的线程以阻塞的方式等待线程分离属性为 `joinable` 的 `thread` 线程运行结束，并获得 `thread` 线程的返回值，返回值的地址保存在 `value_ptr` 里，并释放 `thread` 线程占用的资源。

`pthread_join()` 和 `pthread_detach()` 函数功能类似，都是在线程结束后用来回收线程占用的资源。线程不能等待自己结束，`thread` 线程的分离状态必须是 `joinable`，一个线程只对应一次 `pthread_join()` 调用。分离状态为 `joinable` 的线程仅当有其他线程对其执行了 `pthread_join()` 后，它所占用的资源才会释放。因此为了避免内存泄漏，所有会结束运行的线程，分离状态要么已设为 `detached`，要么使用 `pthread_join()` 来回收其占用的资源。

29.2.4.1 等待线程结束示例代码

以下程序代码会初始化 2 个线程，它们拥有相同的优先级，相同优先级的线程是按照时间片轮转调度。2 个线程属性的分离状态为默认值 `joinable`，线程 1 先开始运行，循环打印 3 次信息后结束。线程 2 调用 `pthread_join()` 阻塞等待线程 1 结束，并回收线程 1 占用的资源，然后线程 2 每隔 2 秒钟会打印一次信息。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
```

```
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    int i;

    for (int i = 0;i < 3;i++) /* 循环打印 3 次信息 */
    {
        printf("thread1 run count: %d\n",i);
        sleep(2); /* 休眠 2 秒 */
    }

    printf("thread1 exited!\n");
    return NULL;
}

/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int count = 0;
    void* thread1_return_value;

    /* 阻塞等待线程 1 运行结束 */
    pthread_join(tid1, NULL);

    /* 线程 2 打印信息开始输出 */
    while(1)
    {
        /* 打印线程计数值输出 */
        printf("thread2 run count: %d\n",count++);
        sleep(2); /* 休眠 2 秒 */
    }

    return NULL;
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    /* 创建线程 1, 属性为默认值, 分离状态为默认值 joinable,
     * 入口函数是 thread1_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid1,NULL,thread1_entry,NULL);
    check_result("thread1 created",result);

    /* 创建线程 2, 属性为默认值, 分离状态为默认值 joinable,
```

```
* 入口函数是 thread2_entry，入口函数参数为 NULL */
result = pthread_create(&tid2,NULL,thread2_entry,NULL);
check_result("thread2 created",result);

return 0;
}
```

29.2.5 退出线程

```
void pthread_exit(void *value_ptr);
```

参数	描述
value_ptr	用户定义的指针，用来存储被等待线程的返回值地址，可由函数 pthread_join() 获取

pthread 线程调用此函数会终止执行，如同进程调用 exit() 函数一样，并返回一个指向线程返回值的指针。线程退出由线程自身发起。

!!! note “注意事项” 若线程的分离状态为 joinable，线程退出后该线程占用的资源并不会被释放，必须调用 pthread_join() 函数释放线程占用的资源。

29.2.5.1 退出线程示例代码

这个程序会初始化 2 个线程，它们拥有相同的优先级，相同优先级的线程是按照时间片轮转调度。2 个线程属性的分离状态为默认值 joinable，线程 1 先开始运行，打印一次信息后休眠 2 秒，之后打印退出信息然后结束运行。线程 2 调用 pthread_join() 阻塞等待线程 1 结束，并回收线程 1 占用的资源，然后线程 2 每隔 2 秒钟会打印一次信息。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值核对函数 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
```

```
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    int count = 0;
    while(1)
    {
        /* 打印线程计数值输出 */
        printf("thread1 run count: %d\n",count++);
        sleep(2);      /* 休眠 2 秒 */
        printf("thread1 will exit!\n");

        pthread_exit(0);      /* 线程 1 主动退出 */
    }
}

/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int count = 0;

    /* 阻塞等待线程 1 运行结束 */
    pthread_join(tid1,NULL);
    /* 线程 2 开始输出打印信息 */
    while(1)
    {
        /* 打印线程计数值输出 */
        printf("thread2 run count: %d\n",count++);
        sleep(2);      /* 休眠 2 秒 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;

    /* 创建线程 1, 属性为默认值, 分离状态为默认值 joinable,
     * 入口函数是 thread1_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid1,NULL,thread1_entry,NULL);
    check_result("thread1 created",result);

    /* 创建线程 2, 属性为默认值, 分离状态为默认值 joinable,
     * 入口函数是 thread2_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid2,NULL,thread2_entry,NULL);
    check_result("thread2 created",result);
}
```

```

    return 0;
}

```

29.3 互斥锁

互斥锁又叫相互排斥的信号量，是一种特殊的二值信号量。互斥锁用来保证共享资源的完整性，保证在任一时刻，只能有一个线程访问该共享资源，线程要访问共享资源，必须先拿到互斥锁，访问完成后需要释放互斥锁。嵌入式的共享资源包括内存、IO、SCI、SPI 等，如果两个线程同时访问共享资源可能会出现问题，因为一个线程可能在另一个线程修改共享资源的过程中使用了该资源，并认为共享资源没有变化。

互斥锁的操作只有两种上锁或解锁，同一时刻只会有一个线程持有某个互斥锁。当有线程持有它时，互斥量处于闭锁状态，由这个线程获得它的所有权。相反，当这个线程释放它时，将对互斥量进行开锁，失去它的所有权。当一个线程持有互斥量时，其他线程将不能够对它进行解锁或持有它。

对互斥锁的主要操作包括：调用 `pthread_mutex_init()` 初始化一个互斥锁，调用 `pthread_mutex_destroy()` 销毁互斥锁，调用 `pthread_mutex_lock()` 对互斥锁上锁，调 `pthread_mutex_unlock()` 对互斥锁解锁。

使用互斥锁会导致一个潜在问题是线程优先级翻转。在 RT-Thread 操作系统中实现的是优先级继承算法。优先级继承是指，提高某个占有某种资源的低优先级线程的优先级，使之与所有等待该资源的线程中优先级最高的那个线程的优先级相等，然后执行，而当这个低优先级线程释放该资源时，优先级重新回到初始设定。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

有关优先级反转的详细介绍请参考《线程间同步》互斥量小节。

29.3.1 互斥锁控制块

每个互斥锁对应一个互斥锁控制块，包含对互斥锁进行的控制的一些信息。创建互斥锁前必须先定义一个 `pthread_mutex_t` 类型的变量，`pthread_mutex_t` 是 `pthread_mutex` 的重定义，`pthread_mutex` 数据结构定义在 `pthread.h` 头文件里，数据结构如下：

```

struct pthread_mutex
{
    pthread_mutexattr_t attr;      /* 互斥锁属性 */
    struct rt_mutex lock;        /* RT-Thread 互斥锁控制块 */
};

typedef struct pthread_mutex pthread_mutex_t;

```

`rt_mutex` 是 RT-Thread 内核里定义的一个数据结构，定义在 `rtdef.h` 头文件里，数据结构如下：

```

struct rt_mutex
{
    struct rt_ipc_object parent;          /* 继承自 ipc_object 类 */
    rt_uint16_t      value;              /* 互斥锁的值 */
    rt_uint8_t       original_priority; /* 持有线程的原始优先级 */
    rt_uint8_t       hold;                /* 互斥锁持有计数 */
    struct rt_thread *owner;             /* 当前拥有互斥锁的线程 */
};

```

```
};

typedef struct rt_mutex* rt_mutex_t;           /* rt_mutex_t 为指向互斥锁结构体的指针
   */
```

29.3.2 初始化互斥锁

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

参数	描述
mutex	互斥锁句柄，不能为 NULL
attr	指向互斥锁属性的指针，若该指针 NULL，则使用默认的属性。
返回	—
0	成功
EINVAL	参数无效

此函数会初始化 mutex 互斥锁，并根据 attr 指向的互斥锁属性对象设置 mutex 属性，成功初始化后互斥锁处于未上锁状态，线程可以获取，此函数是对 rt_mutex_init() 函数的封装。

除了调用 pthread_mutex_init() 函数创建一个互斥锁，还可以用宏 PTHREAD_MUTEX_INITIALIZER 来静态初始化互斥锁，方法：pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER（结构体常量），等同于调用 pthread_mutex_init() 时 attr 指定为 NULL。

关于互斥锁属性及相关函数会在线程高级编程一章里有详细介绍，一般情况下采用默认属性就可以。

29.3.3 销毁互斥锁

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

参数	描述
mutex	互斥锁句柄，不能为 NULL
返回	—
0	成功
EINVAL	mutex 为空或者 mutex 已经被销毁过
EBUSY	互斥锁正在被使用

此函数会销毁 mutex 互斥锁。销毁后互斥锁 mutex 处于未初始化状态。销毁以后互斥锁的属性和控制块参数将不在有效，但可以调用 pthread_mutex_init() 对销毁后的互斥锁重新初始化。但不需要销毁使用宏 PTHREAD_MUTEX_INITIALIZER 静态初始化的互斥锁。

当确定互斥锁没有被锁住，且没有线程阻塞在该互斥锁上，才可以销毁该互斥锁。

29.3.4 阻塞方式对互斥锁上锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

参数	描述
mutex	互斥锁句柄，不能为 NULL
返回	—
0	成功
EINVAL	参数无效
EDEADLK	互斥锁 mutex 不为嵌套锁的情况下线程重复调用此函数

此函数对 mutex 互斥锁上锁，此函数是对 rt_mutex_take() 函数的封装。如果互斥锁 mutex 还没有被上锁，那么申请该互斥锁的线程将成功对该互斥锁上锁。如果互斥锁 mutex 已经被当前线程上锁，且互斥锁类型为嵌套锁，则该互斥锁的持有计数加 1，当前线程也不会挂起等待（死锁），但线程必须对应相同次数的解锁。如果互斥锁 mutex 被其他线程上锁持有，则当前线程将被阻塞，一直到其他线程对该互斥锁解锁后，等待该互斥锁的线程将按照先进先出的原则获取互斥锁。

29.3.5 非阻塞方式对互斥锁上锁

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

参数	描述
mutex	互斥锁句柄，不能为 NULL
返回	—
0	成功
EINVAL	参数无效
EDEADLK	互斥锁 mutex 不为嵌套锁的情况下线程重复调用此函数
EBUSY	互斥锁 mutex 已经被其他线程上锁

此函数是 pthread_mutex_lock() 函数的非阻塞版本。区别在于如果互斥锁 mutex 已经被上锁，线程不会被阻塞，而是马上返回错误码。

29.3.6 解锁互斥锁

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

参数	描述
mutex	互斥锁句柄，不能为 NULL
返回	—
0	成功
EINVAL	参数无效
EPERM	互斥锁 mutex 不为嵌套锁的情况下线程重复调用此函数
EBUSY	解锁其他线程持有的类型为检错锁的互斥锁

调用此函数给 mutex 互斥锁解锁，是对 rt_mutex_release() 函数的封装。当线程完成共享资源的访问后，应尽快释放占有的互斥锁，使得其他线程能及时获取该互斥锁。只有已经拥有互斥锁的线程才能释放它，每释放一次该互斥锁，它的持有计数就减 1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），互斥锁才变为可用，等待在该互斥锁上的线程将按先进先出方式被唤醒。如果线程的运行优先级被互斥锁提升，那么当互斥锁被释放后，线程恢复为持有互斥锁前的优先级。

29.3.7 互斥锁示例代码

这个程序会初始化 2 个线程，它们拥有相同的优先级，2 个线程都会调用同一个 printer() 函数输出自己的字符串，printer() 函数每次只输出一个字符，之后休眠 1 秒，调用 printer() 函数的线程同样也休眠。如果不使用互斥锁，线程 1 打印了一个字符，休眠后执行线程 2，线程 2 打印一个字符，这样就不能完整的打印线程 1 和线程 2 的字符串，打印出的字符串是混乱的。如果使用了互斥锁保护 2 个线程共享的打印函数 printer()，线程 1 拿到互斥锁后执行 printer() 打印函数打印一个字符，之后休眠 1 秒，这是切换到线程 2，因为互斥锁已经被线程 1 上锁，线程 2 将阻塞，直到线程 1 的字符串打印完整后主动释放互斥锁后线程 2 才会被唤醒。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;
/* 互斥锁控制块 */
static pthread_mutex_t mutex;
/* 线程共享的打印函数 */
static void printer(char* str)
{
    while(*str != 0)
    {
        putchar(*str); /* 输出一个字符 */
        str++;
        sleep(1); /* 休眠 1 秒 */
    }
    printf("\n");
}
```

```
/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程入口 */
static void* thread1_entry(void* parameter)
{
    char* str = "thread1 hello RT-Thread";
    while (1)
    {
        pthread_mutex_lock(&mutex);      /* 互斥锁上锁 */

        printer(str); /* 访问共享打印函数 */

        pthread_mutex_unlock(&mutex); /* 访问完成后解锁 */

        sleep(2); /* 休眠 2 秒 */
    }
}

static void* thread2_entry(void* parameter)
{
    char* str = "thread2 hi world";
    while (1)
    {
        pthread_mutex_lock(&mutex); /* 互斥锁上锁 */

        printer(str); /* 访问共享打印函数 */

        pthread_mutex_unlock(&mutex); /* 访问完成后解锁 */

        sleep(2); /* 休眠 2 秒 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    /* 初始化一个互斥锁 */
    pthread_mutex_init(&mutex,NULL);

    /* 创建线程 1, 线程入口是 thread1_entry, 属性参数为 NULL 选择默认值, 入口参数是 */
}
```

```

    NULL*/
result = pthread_create(&tid1,NULL,thread1_entry,NULL);
check_result("thread1 created",result);

/* 创建线程 2, 线程入口是 thread2_entry, 属性参数为 NULL 选择默认值, 入口参数是
NULL*/
result = pthread_create(&tid2,NULL,thread2_entry,NULL);
check_result("thread2 created",result);

return 0;
}

```

29.4 条件变量

条件变量其实就是一个信号量，用于线程间同步。条件变量用来阻塞一个线程，当条件满足时向阻塞的线程发送一个条件，阻塞线程就被唤醒，条件变量需要和互斥锁配合使用，互斥锁用来保护共享数据。

条件变量可以用来通知共享数据状态。比如一个处理共享资源队列的线程发现队列为空，则此线程只能等待，直到有一个节点被添加到队列中，添加后在发一个条件变量信号激活等待线程。

条件变量的主要操作包括：调用 `pthread_cond_init()` 对条件变量初始化，调用 `pthread_cond_destroy()` 销毁一个条件变量，调用 `pthread_cond_wait()` 等待一个条件变量，调用 `pthread_cond_signal()` 发送一个条件变量。

29.4.1 条件变量控制块

每个条件变量对应一个条件变量控制块，包括对条件变量进行操作的一些信息。初始化一个条件变量前需要先定义一个 `pthread_cond_t` 条件变量控制块。`pthread_cond_t` 是 `pthread_cond` 结构体类型的重定义，定义在 `pthread.h` 头文件里。

```

struct pthread_cond
{
    pthread_condattr_t attr;          /* 条件变量属性 */
    struct rt_semaphore sem;          /* RT-Thread 信号量控制块 */
};

typedef struct pthread_cond pthread_cond_t;

```

`rt_semaphore` 是 RT-Thread 内核里定义的一个数据结构，是信号量控制块，定义在 `rtdef.h` 头文件里

```

struct rt_semaphore
{
    struct rt_ipc_object parent; /* 继承自 ipc_object 类 */
    rt_uint16_t value;        /* 信号量的值 */
};

```

29.4.2 初始化条件变量

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

参数	描述
cond	条件变量句柄，不能为 NULL
attr	指向条件变量属性的指针，若为 NULL 则使用默认属性值
返回	—
0	成功
EINVAL	参数无效

此函数会初始化 cond 条件变量，并根据 attr 指向的条件变量属性设置其属性，此函数是对 rt_sem_init() 函数的一个封装，基于信号量实现。初始化成功后条件变量处于不可用状态。

还可以用宏 PTHREAD_COND_INITIALIZER 静态初始化一个条件变量，方法：`pthread_cond_t cond = PTHREAD_COND_INITIALIZER`（结构体常量），等同于调用 `pthread_cond_init()` 时 attr 指定为 NULL。

attr 一般设置 NULL 使用默认值即可，具体会在线程高级编程一章介绍。

29.4.3 销毁条件变量

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

参数	描述
cond	条件变量句柄，不能为 NULL
返回	—
0	成功
EINVAL	参数无效
EPERM	互斥锁 mutex 不为嵌套锁的情况下线程重复调用此函数
EBUSY	条件变量正在被使用

此函数会销毁 cond 条件变量，销毁后 cond 处于未初始化状态。销毁之后条件变量的属性及控制块参数将不再有效，但可以调用 `pthread_cond_init()` 或者静态方式重新初始化。

销毁条件变量前需要确定没有线程被阻塞在该条件变量上，也不会等待获取、发信号或者广播。

29.4.4 阻塞方式获取条件变量

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

参数	描述
cond	条件变量句柄, 不能为 NULL
mutex	指向互斥锁控制块的指针, 不能为 NULL
返回	—
0	成功
EINVAL	参数无效

此函数会以阻塞方式获取 **cond** 条件变量。线程等待条件变量前需要先将 **mutex** 互斥锁锁住，此函数首先判断条件变量是否可用，如果不可用则初始化一个条件变量，之后解锁 **mutex** 互斥锁，然后尝试获取一个信号量，当信号量值大于零时，表明信号量可用，线程将获得信号量，也就获得该条件变量，相应的信号量值会减 1。如果信号量的值等于零，表明信号量不可用，线程将阻塞直到信号量可用，之后将对 **mutex** 互斥锁再次上锁。

29.4.5 指定阻塞时间获取条件变量

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

参数	描述
cond	条件变量句柄, 不能为 NULL
mutex	指向互斥锁控制块的指针, 不能为 NULL
abstime	指定的等待时间, 单位是操作系统时钟节拍 (OS Tick)
返回	—
0	成功
EINVAL	参数无效
EPERM	互斥锁 mutex 不为嵌套锁的情况下线程重复调用此函数
ETIMEDOUT	超时

此函数和 `pthread_cond_wait()` 函数唯一的差别在于，如果条件变量不可用，线程将被阻塞 `abstime` 时长，超时后函数将直接返回 `ETIMEDOUT` 错误码，线程将会被唤醒进入就绪态。

29.4.6 发送满足条件信号量

```
int pthread_cond_signal(pthread_cond_t *cond);
```

参数	描述
cond	条件变量句柄, 不能为 NULL
返回	—
0	成功

此函数会发送一个信号且只唤醒一个等待 cond 条件变量的线程, 是对 rt_sem_release() 函数的封装, 也就是发送一个信号量。当信号量的值等于零, 并且有线程等待这个信号量时, 将唤醒等待在该信号量线程队列中的第一个线程, 由它获取信号量。否则将把信号量的值加 1。

29.4.7 广播

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

参数	描述
cond	条件变量句柄, 不能为 NULL
返回	—
0	成功
EINVAL	参数无效

调用此函数将唤醒所有等待 cond 条件变量的线程。

29.4.8 条件变量示例代码

这个程序是一个生产者消费者模型, 有一个生产者线程, 一个消费者线程, 它们拥有相同的优先级。生产者每隔 2 秒会生产一个数字, 放到 head 指向的链表里面, 之后调用 pthread_cond_signal() 给消费者线程发信号, 通知消费者线程链表里面有数据。消费者线程会调用 pthread_cond_wait() 等待生产者线程发送信号。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* 静态方式初始化一个互斥锁和一个条件变量 */
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* 指向线程控制块的指针 */

```

```
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 生产者生产的结构体数据，存放在链表里 */
struct node
{
    int n_number;
    struct node* n_next;
};
struct node* head = NULL; /* 链表头，是共享资源 */

/* 消费者线程入口函数 */
static void* consumer(void* parameter)
{
    struct node* p_node = NULL;

    pthread_mutex_lock(&mutex); /* 对互斥锁上锁 */

    while (1)
    {
        while (head == NULL) /* 判断链表里是否有元素 */
        {
            pthread_cond_wait(&cond,&mutex); /* 尝试获取条件变量 */
        }
        /*
         * pthread_cond_wait() 会先对 mutex 解锁，
         * 然后阻塞在等待队列，直到获取条件变量被唤醒，
         * 被唤醒后，该线程会再次对 mutex 上锁，成功进入临界区。
         */

        p_node = head; /* 拿到资源 */
        head = head->n_next; /* 头指针指向下一个资源 */
        /* 打印输出 */
        printf("consume %d\n",p_node->n_number);

        free(p_node); /* 拿到资源后释放节点占用的内存 */
    }
}
```

```
    }
    pthread_mutex_unlock(&mutex);      /* 释放互斥锁 */
    return 0;
}

/* 生产者线程入口函数 */
static void* product(void* parameter)
{
    int count = 0;
    struct node *p_node;

    while(1)
    {
        /* 动态分配一块结构体内存 */
        p_node = (struct node*)malloc(sizeof(struct node));
        if (p_node != NULL)
        {
            p_node->n_number = count++;
            pthread_mutex_lock(&mutex);      /* 需要操作 head 这个临界资源，先加锁 */

            p_node->n_next = head;
            head = p_node;      /* 往链表头插入数据 */

            pthread_mutex_unlock(&mutex);      /* 解锁 */
            printf("produce %d\n",p_node->n_number);

            pthread_cond_signal(&cond);      /* 发信号唤醒一个线程 */

            sleep(2);      /* 休眠 2 秒 */
        }
        else
        {
            printf("product malloc node failed!\n");
            break;
        }
    }
}

int rt_application_init()
{
    int result;

    /* 创建生产者线程，属性为默认值，入口函数是 product，入口函数参数为 NULL*/
    result = pthread_create(&tid1,NULL,product,NULL);
    check_result("product thread created",result);

    /* 创建消费者线程，属性为默认值，入口函数是 consumer，入口函数参数是 NULL */
    result = pthread_create(&tid2,NULL,consumer,NULL);
    check_result("consumer thread created",result);
```

```
    return 0;
}
```

29.5 读写锁

读写锁也称为多读者单写者锁。读写锁把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。同一时间只能有一个线程可以占有写模式的读写锁，但是可以有多个线程同时占有读模式的读写锁。读写锁适合于对数据结构的读次数比写次数多得多的情况，因为读模式锁定时可以共享，写模式锁定时意味着独占。

读写锁通常是基于互斥锁和条件变量实现的。一个线程可以对一个读写锁进行多次读写锁定，同样必须有对应次数的解锁。

读写锁的主要操作包括：调用 `pthread_rwlock_init()` 初始化一个读写锁，写线程调用 `pthread_rwlock_wrlock()` 对读写锁写锁定，读线程调用 `pthread_rwlock_rdlock()` 对读写锁读锁定，当不需要使用此读写锁时调用 `pthread_rwlock_destroy()` 销毁读写锁。

29.5.1 读写锁控制块

每个读写锁对应一个读写锁控制块，包括对读写锁进行操作的一些信息。`pthread_rwlock_t` 是 `pthread_rwlock` 数据结构的重定义，定义在 `pthread.h` 头文件里。在创建一个读写锁之前需要先定义一个 `pthread_rwlock_t` 类型的数据结构。

```
struct pthread_rwlock
{
    pthread_rwlockattr_t attr;      /* 读写锁属性 */
    pthread_mutex_t rw_mutex;       /* 互斥锁 */
    pthread_cond_t rw_condreaders;   /* 条件变量，供读者线程使用 */
    pthread_cond_t rw_condwriters;  /* 条件变量，供写者线程使用 */
    int rw_nwaitreaders;           /* 读者线程等待计数 */
    int rw_nwaitwriters;           /* 写者线程等待计数 */
    /* 读写锁值，值为 0：未上锁，值为 -1：被写者线程锁定，大于 0 值：被读者线程锁定
       数量 */
    int rw_refcount;
};

typedef struct pthread_rwlock pthread_rwlock_t;          /* 类型重定义 */
```

29.5.2 初始化读写锁初始化

```
int pthread_rwlock_init (pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
```

参数	描述
----	----

rwlock	读写锁句柄，不能为 NULL
--------	----------------

参数	描述
attr	指向读写锁属性的指针, RT-Thread 不使用此变量
返回	—
0	成功
EINVAL	参数无效

此函数会初始化一个 `rwlock` 读写锁。此函数使用默认值初始化读写锁控制块的信号量和条件变量，相关计数参数初始为 0 值。初始化后的读写锁处于未上锁状态。

还可以使用宏 `PTHREAD_RWLOCK_INITIALIZER` 来静态初始化读写锁，方法：`pthread_rwlock_t mutex = PTHREAD_RWLOCK_INITIALIZER`（结构体常量），等同于调用 `pthread_rwlock_init()` 时 attr 指定为 `NULL`。

`attr` 一般设置 `NULL` 使用默认值即可，具体会在线程高级编程一章介绍。

29.5.3 销毁读写锁

```
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 <code>NULL</code>
返回	—
0	成功
EINVAL	参数无效
EBUSY	读写锁目前正在被使用或者有线程等待该读写锁
EDEADLK	死锁

此函数会销毁一个 `rwlock` 读写锁，对应的会销毁读写锁里的互斥锁和条件变量。销毁之后读写锁的属性及控制块参数将不在有效，但可以调用 `pthread_rwlock_init()` 或者静态方式重新初始化读写锁。

29.5.4 读写锁读锁定

29.5.4.1 阻塞方式对读写锁读锁定

```
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 <code>NULL</code>

参数	描述
返回	—
0	成功
EINVAL	参数无效
EDEADLK	死锁

读者线程可以调用此函数来对 `rwlock` 读写锁进行读锁定。如果读写锁没有被写锁定并且没有写者线程阻塞在该读写锁上，读写线程将成功获取该读写锁。如果读写锁已经被写锁定，读者线程将会阻塞，直到写锁定该读写锁的线程解锁。

29.5.4.2 非阻塞方式对读写锁读锁定

```
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 NULL
返回	—
0	成功
EINVAL	参数无效
EBUSY	读写锁目前正在被使用或者有线程等待该读写锁
EDEADLK	死锁

此函数和 `pthread_rwlock_rdlock()` 函数的不同在于，如果读写锁已经被写锁定，读者线程不会被阻塞，而是返回一个错误码 `EBUSY`。

29.5.4.3 指定阻塞时间对读写锁读锁定

```
int pthread_rwlock_timedrdlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abstime);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 NULL
<code>abstime</code>	指定的等待时间，单位是操作系统时钟节拍（OS Tick）
返回	—
0	成功
EINVAL	参数无效

参数	描述
ETIMEDOUT	超时
EDEADLK	死锁

此函数和 `pthread_rwlock_rdlock()` 函数的不同在于，如果读写锁已经被写锁定，读者线程将会阻塞指定的 `abstime` 时长，超时后函数将返回错误码 `ETIMEDOUT`，线程将被唤醒进入就绪态。

29.5.5 读写锁写锁定

29.5.5.1 阻塞方式对读写锁写锁定

```
int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 <code>NULL</code>
返回	—
0	成功
<code>EINVAL</code>	参数无效
<code>EDEADLK</code>	死锁

写者线程调用此函数对 `rwlock` 读写锁进行写锁定。写锁定读写锁类似互斥量，同一时刻只能有一个线程写锁定读写锁。如果没有线程锁定该读写锁，即读写锁值为 0，调用此函数的写者线程将会写锁定读写锁，其他线程此时都不能获取读写锁，如果已经有线程锁定该读写锁，即读写锁值不为 0，则写线程将被阻塞，直到读写锁解锁。

29.5.5.2 非阻塞方式写锁定读写锁

```
int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 <code>NULL</code>
返回	—
0	成功
<code>EINVAL</code>	参数无效
<code>EBUSY</code>	读写锁目前被写锁定或者有写线程阻塞在该读写锁上
<code>EDEADLK</code>	死锁

此函数和 `pthread_rwlock_wrlock()` 函数唯一的不同在于，如果有线程锁定该读写锁，即读写锁值不为 0，则调用该函数的写者线程会直接返回一个错误代码，线程不会被阻塞。

29.5.5.3 指定阻塞时长写锁定读写锁

```
int pthread_rwlock_timedwrlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abstime);
```

参数	描述
<code>rwlock abstime</code>	读写锁句柄，不能为 NULL 指定的等待时间，单位是操作系统时钟节拍 (OS Tick)
返回	—
0	成功
<code>EINVAL</code>	参数无效
<code>ETIMEDOUT</code>	超时
<code>EDEADLK</code>	死锁

此函数和 `pthread_rwlock_wrlock()` 函数唯一的不同在于，如果有线程锁定该读写锁，即读写锁值不为 0，调用线程阻塞指定的 `abstime` 时长，超时后函数将返回错误码 `ETIMEDOUT`，线程将会被唤醒进入就绪态。

29.5.6 读写锁解锁

```
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

参数	描述
<code>rwlock</code>	读写锁句柄，不能为 NULL
返回	—
0	成功
<code>EINVAL</code>	参数无效
<code>EDEADLK</code>	死锁

此函数可以对 `rwlock` 读写锁解锁。线程对同一个读写锁加锁多次，必须有同样次数的解锁，若解锁后有多个线程等待对读写锁进行锁定，系统将按照先进先出的规则激活等待的线程。

29.5.7 读写锁示例代码

这个程序有 2 个读者线程，一个写着线程。2 个读者线程先对读写锁读锁定，之后休眠 2 秒，这是其他的读者线程还是可以对该读写锁读锁定，然后读取共享数据。

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t reader1;
static pthread_t reader2;
static pthread_t writer1;
/* 共享数据 book */
static int book = 0;
/* 读写锁 */
static pthread_rwlock_t rwlock;
/* 函数结果检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}
/* 线程入口 */
static void* reader1_entry(void* parameter)
{
    while (1)
    {
        pthread_rwlock_rdlock(&rwlock); /* 尝试读锁定该读写锁 */

        printf("reader1 read book value is %d\n",book);
        sleep(2); /* 线程休眠 2 秒，切换到其他线程运行 */

        pthread_rwlock_unlock(&rwlock); /* 线程运行后对读写锁解锁 */
    }
}
static void* reader2_entry(void* parameter)
{
    while (1)
    {
        pthread_rwlock_rdlock(&rwlock); /* 尝试读锁定该读写锁 */
```

```
printf("reader2 read book value is %d\n",book);
sleep(2); /* 线程休眠 2 秒，切换到其他线程运行 */

pthread_rwlock_unlock(&rwlock); /* 线程运行后对读写锁解锁 */
}

}

static void* writer1_entry(void* parameter)
{
    while (1)
    {
        pthread_rwlock_wrlock(&rwlock); /* 尝试写锁定该读写锁 */

        book++;
        printf("writer1 write book value is %d\n",book);

        pthread_rwlock_unlock(&rwlock); /* 对读写锁解锁 */

        sleep(2); /* 线程休眠 2 秒，切换到其他线程运行 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    /* 默认属性初始化读写锁 */
    pthread_rwlock_init(&rwlock,NULL);

    /* 创建 reader1 线程，线程入口是 reader1_entry，线程属性为默认值，入口参数为 NULL*/
    result = pthread_create(&reader1,NULL,reader1_entry,NULL);
    check_result("reader1 created",result);

    /* 创建 reader2 线程，线程入口是 reader2_entry，线程属性为默认值，入口参数为 NULL*/
    result = pthread_create(&reader2,NULL,reader2_entry,NULL);
    check_result("reader2 created",result);

    /* 创建 writer1 线程，线程入口是 writer1_entry，线程属性为，入口参数为 NULL*/
    result = pthread_create(&writer1,NULL,writer1_entry,NULL);
    check_result("writer1 created",result);

    return 0;
}
```

29.6 屏障

屏障是多线程同步的一种方法。**barrier** 意为屏障或者栏杆，把先后到达的多个线程挡在同一栏杆前，直到所有线程到齐，然后撤下栏杆同时放行。先到达的线程将会阻塞，等到所有调用 `pthread_barrier_wait()` 函数的线程（数量等于屏障初始化时指定的 `count`）都到达后，这些线程才会由阻塞状态进入就绪状态再次参与系统调度。

屏障是基于条件变量和互斥锁实现的。主要操作包括：调用 `pthread_barrier_init()` 初始化一个屏障，其他线程调用 `pthread_barrier_wait()`，所有线程到期后线程唤醒进入准备状态，屏障不在使用调用 `pthread_barrier_destroy()` 销毁一个屏障。

29.6.1 屏障控制块

创建一个屏障前需要先定义一个 `pthread_barrier_t` 屏障控制块。`pthread_barrier_t` 是 `pthread_barrier` 结构体类型的重定义，定义在 `pthread.h` 头文件里。

```
struct pthread_barrier
{
    int count;      /* 指定的等待线程个数 */
    pthread_cond_t cond;        /* 条件变量 */
    pthread_mutex_t mutex;      /* 互斥锁 */
};

typedef struct pthread_barrier pthread_barrier_t;
```

29.6.2 创建屏障

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                        const pthread_barrierattr_t *attr,
                        unsigned count);
```

参数	描述
attr	指向屏障属性的指针，传入 <code>NULL</code> ，则使用默认值，非 <code>NULL</code> 必须使用 <code>PTHREAD_PROCESS_PRIVATE</code>
barrier	屏障句柄
count	指定的等待线程个数
返回	—
0	成功
EINVAL	参数无效

此函数会创建一个 `barrier` 屏障，并根据默认的参数对屏障控制块的条件变量和互斥锁初始化，初始化后指定的等待线程个数为 `count` 个，必须对应 `count` 个线程调用 `pthread_barrier_wait()`。

`attr` 一般设置 `NULL` 使用默认值即可，具体会在线程高级编程一章介绍。

29.6.3 销毁屏障

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

参数	描述
barrier	屏障句柄
返回	—
0	成功
EINVAL	参数无效

此函数会销毁一个 barrier 屏障。销毁之后屏障的属性及控制块参数将不在有效，但可以调用 pthread_barrier_init() 重新初始化。

29.6.4 等待屏障

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

参数	描述
barrier	屏障句柄
返回	—
0	成功
EINVAL	参数无效

此函数同步等待在 barrier 前的线程，由每个线程主动调用，若屏障等待线程个数 count 不为 0，count 将减 1，若减 1 后 count 为 0，表明所有线程都已经到达栏杆前，所有到达的线程将被唤醒重新进入就绪状态，参与系统调度。若减一后 count 不为 0，表明还有线程没有到达屏障，调用的线程将阻塞直到所有线程到达屏障。

29.6.5 屏障示例代码

此程序会创建 3 个线程，初始化一个屏障，屏障等待线程数初始化为 3。3 个线程都会调用 pthread_barrier_wait() 等待在屏障前，当 3 个线程都到齐后，3 个线程进入就绪态，之后会每隔 2 秒打印输出计数信息。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
```

```
static pthread_t tid2;
static pthread_t tid3;
/* 屏障控制块 */
static pthread_barrier_t barrier;
/* 函数返回值检查函数 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}
/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    int count = 0;

    printf("thread1 have arrived the barrier!\n");
    pthread_barrier_wait(&barrier);      /* 到达屏障，并等待其他线程到达 */

    while (1)
    {
        /* 打印线程计数值输出 */
        printf("thread1 count: %d\n",count ++);

        /* 休眠 2 秒 */
        sleep(2);
    }
}
/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int count = 0;

    printf("thread2 have arrived the barrier!\n");
    pthread_barrier_wait(&barrier);

    while (1)
    {
        /* 打印线程计数值输出 */
        printf("thread2 count: %d\n",count ++);

        /* 休眠 2 秒 */
        sleep(2);
    }
}
```

```
}

/* 线程 3 入口函数 */
static void* thread3_entry(void* parameter)
{
    int count = 0;

    printf("thread3 have arrived the barrier!\n");
    pthread_barrier_wait(&barrier);

    while (1)
    {
        /* 打印线程计数值输出 */
        printf("thread3 count: %d\n", count++);
        sleep(2);
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    pthread_barrier_init(&barrier, NULL, 3);

    /* 创建线程 1, 线程入口是 thread1_entry, 属性参数设为 NULL 选择默认值, 入口参数
       为 NULL*/
    result = pthread_create(&tid1, NULL, thread1_entry, NULL);
    check_result("thread1 created", result);

    /* 创建线程 2, 线程入口是 thread2_entry, 属性参数设为 NULL 选择默认值, 入口参数
       为 NULL*/
    result = pthread_create(&tid2, NULL, thread2_entry, NULL);
    check_result("thread2 created", result);

    /* 创建线程 3, 线程入口是 thread3_entry, 属性参数设为 NULL 选择默认值, 入口参数
       为 NULL*/
    result = pthread_create(&tid3, NULL, thread3_entry, NULL);
    check_result("thread3 created", result);
}
```

29.7 信号量

信号量可以用于进程与进程之间，或者进程内线程之间的通信。每个信号量都有一个不会小于 0 的信号量值，对应信号量的可用数量。调用 `sem_init()` 或者 `sem_open()` 给信号量值赋初值，调用 `sem_post()` 函数可以让信号量值加 1，调用 `sem_wait()` 可以让信号量值减 1，如果当前信号量为 0，调用 `sem_wait()` 的线程被挂起在该信号量的等待队列上，直到信号量值大于 0，处于可用状态。

根据信号量的值（代表可用资源的数目）的不同，POSIX 信号量可以分为：

- 二值信号量：信号量的值只有 0 和 1，初始值指定为 1。这和互斥锁一样，若资源被锁住，信号量的值为 0，若资源可用，则信号量的值为 1。相当于只有一把钥匙，线程拿到钥匙后，完成了对共享资源的访问后需要解锁，把钥匙再放回去，给其他需要此钥匙的线程使用。使用方法和互斥锁一样，等待信号量函数必须和发送信号量函数成对使用，不能单独使用，必须先等待后发送。
- 计数信号量：信号量的值在 0 到一个大于 1 的限制值（POSIX 指出系统的最大限制值至少要为 32767）。该计数表示可用信号量个数。此时，发送信号量函数可以被单独调用发送信号量，相当于有多把钥匙，线程拿到一把钥匙就消耗了一把，使用过的钥匙不必在放回去。

POSIX 信号量又分为有名信号量和无名信号量：

- 有名信号量：其值保存在文件中，一般用于进程间同步或互斥。
- 无名信号量：其值保存在内存中，一般用于线程间同步或互斥。

RT-Thread 操作系统的 POSIX 信号量主要是基于 RT-Thread 内核信号量的一个封装，主要还是用于系统内线程间的通讯。使用方式和 RT-Thread 内核的信号量差不多。

29.7.1 信号量控制块

每个信号量对应一个信号量控制块，创建一个信号量前需要先定义一个 `sem_t` 信号量控制块。`sem_t` 是 `posix_sem` 结构体类型的重定义，定义在 `semaphore.h` 头文件里。

```
struct posix_sem
{
    rt_uint16_t refcount;
    rt_uint8_t unlinked;
    rt_uint8_t unnamed;
    rt_sem_t sem;      /* RT-Thread 信号量 */
    struct posix_sem* next;     /* 指向下一个信号量控制块 */
};

typedef struct posix_sem sem_t;
```

`rt_sem_t` 是 RT-Thread 信号量控制块，定义在 `rtdef.h` 头文件里。

```
struct rt_semaphore
{
    struct rt_ipc_object parent; /* 继承自 ipc_object 类 */
    rt_uint16_t value;        /* 信号量的值 */
};

/* rt_sem_t 是指向 semaphore 结构体的指针类型 */
typedef struct rt_semaphore* rt_sem_t;
```

29.7.2 无名信号量

无名信号量的值保存在内存中，一般用于线程间同步或互斥。在使用之前，必须先调用 `sem_init()` 初始化。

29.7.2.1 初始化无名信号量

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

参数	描述
sem	信号量句柄
value	信号量初始值，表示信号量资源的可用数量
pshared	RT-Thread 未实现参数
返回	—
0	成功
-1	失败

此函数初始化一个无名信号量 `sem`，根据给定的或默认的参数对信号量相关数据结构进行初始化，并把信号量放入信号量链表里。初始化后信号量值为给定的初始值 `value`。此函数是对 `rt_sem_create()` 函数的封装。

29.7.2.2 销毁无名信号量

```
int sem_destroy(sem_t *sem);
```

参数	描述
sem	信号量句柄
返回	—
0	成功
-1	失败

此函数会销毁一个无名信号量 `sem`，并释放信号量占用的资源。

29.7.3 有名信号量

有名信号量，其值保存在文件中，一般用于进程间同步或互斥。两个进程可以操作相同名称的有名信号量。RT-Thread 操作系统中的有名信号量实现和无名信号量差不多，都是设计用于线程间的通信，使用方法也类似。

29.7.3.1 创建或打开有名信号量

```
sem_t *sem_open(const char *name, int oflag, ...);
```

参数	描述
name	信号量名称
oflag	信号量的打开方式
返回	—
信号量句柄	成功
NULL	失败

此函数会根据信号量名字 name 创建一个新的信号量或者打开一个已经存在的信号量。Oflag 的可选值有 0、O_CREAT 或 O_CREAT|O_EXCL。如果 Ofag 设置为 O_CREAT 则会创建一个新的信号量。如果 Ofag 设置 O_CREAT|O_EXCL，如果信号量已经存在则会返回 NULL，如果不存在则会创建一个新的信号量。如果 Ofag 设置为 0，信号量不存在则会返回 NULL。

29.7.3.2 分离有名信号量

```
int sem_unlink(const char *name);
```

参数	描述
name	信号量名称
返回	—
0	成功
-1	失败，信号量不存在

此函数会根据信号量名称 name 查找该信号量，若信号量存在，则将该信号量标记为分离状态。之后检查引用计数，若值为 0，则立即删除信号量，若值不为 0，则等到所有持有该信号量的线程关闭信号量之后才会删除。

29.7.3.3 关闭有名信号量

```
int sem_close(sem_t *sem);
```

参数	描述
sem	信号量句柄
返回	—
0	成功
-1	失败

当一个线程终止时，会对其占用的信号量执行此关闭操作。不论线程是自愿终止还是非自愿终止都会执行这个关闭操作，相当于是信号量的持有计数减 1。若减 1 后持有计数为 0 且信号量已经处于分离状态，则会删除 sem 信号量并释放其占有的资源。

29.7.4 获取信号量值

```
int sem_getvalue(sem_t *sem, int *sval);
```

参数	描述
sem	信号量句柄，不能为 NULL
sval	保存获取的信号量值地址，不能为 NULL
返回	—
0	成功
-1	失败

此函数可以获取 sem 信号量的值，并保存在 sval 指向的内存里，可以知道信号量的资源数量。

29.7.5 阻塞方式等待信号量

```
int sem_wait(sem_t *sem);
```

参数	描述
sem	信号量句柄，不能为 NULL
返回	—
0	成功
-1	失败

线程调用此函数获取信号量，是 rt_sem_take(sem, RT_WAITING_FOREVER) 函数的封装。若信号量值大于零，表明信号量可用，线程获得信号量，信号量值减 1。若信号量值等于 0，表明信号量不可用，线程阻塞进入挂起状态，并按照先进先出的方式排队等待，直到信号量可用。

29.7.6 非阻塞方式获取信号量

```
int sem_trywait(sem_t *sem);
```

参数	描述
sem	信号量句柄，不能为 NULL

参数	描述
返回	—
0	成功
-1	失败

此函数是 `sem_wait()` 函数的非阻塞版，是 `rt_sem_take(sem,0)` 函数的封装。当信号量不可用时，线程不会阻塞，而是直接返回。

29.7.7 指定阻塞时间等待信号量

```
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

参数	描述
sem	信号量句柄，不能为 NULL
abs_timeout	指定的等待时间，单位是操作系统时钟节拍（OS Tick）
返回	—
0	成功
-1	失败

此函数和 `sem_wait()` 函数的区别在于，若信号量不可用，线程将阻塞 `abs_timeout` 时长，超时后函数返回 -1，线程将被唤醒由阻塞态进入就绪态。

29.7.8 发送信号量

```
int sem_post(sem_t *sem);
```

参数	描述
sem	信号量句柄，不能为 NULL
返回	—
0	成功
-1	失败

此函数将释放一个 `sem` 信号量，是 `rt_sem_release()` 函数的封装。若等待该信号量的线程队列不为空，表明有线程在等待该信号量，第一个等待该信号量的线程将由挂起状态切换到就绪状态，等待系统调度。若没有线程等待该信号量，该信号量值将加 1。

29.7.9 无名信号量使用示例代码

信号量使用的典型案例是生产者消费者模型。一个生产者线程和一个消费者线程对同一块内存进行操作，生产者往共享内存填充数据，消费者从共享内存读取数据。

此程序会创建 2 个线程，2 个信号量，一个信号量表示共享数据为空状态，一个信号量表示共享数据不为空状态，一个互斥锁用于保护共享资源。生产者线程生产好数据后会给消费者发送一个 `full_sem` 信号量，通知消费者线程有数据可用，休眠 2 秒后会等待消费者线程发送的 `empty_sem` 信号量。消费者线程等到生产者发送的 `full_sem` 后会处理共享数据，处理完后会给生产者线程发送 `empty_sem` 信号量。程序会这样一直循环。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>

/* 静态方式初始化一个互斥锁用于保护共享资源 */
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
/* 2 个信号量控制块，一个表示资源空信号，一个表示资源满信号 */
static sem_t empty_sem,full_sem;

/* 指向线程控制块的指针 */
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 生产者生产的结构体数据，存放在链表里 */
struct node
{
    int n_number;
    struct node* n_next;
};

struct node* head = NULL; /* 链表头，是共享资源 */

/* 消费者线程入口函数 */
static void* consumer(void* parameter)
```

```
{  
    struct node* p_node = NULL;  
  
    while (1)  
    {  
        sem_wait(&full_sem);  
        pthread_mutex_lock(&mutex); /* 对互斥锁上锁， */  
  
        while (head != NULL) /* 判断链表里是否有元素 */  
        {  
            p_node = head; /* 拿到资源 */  
            head = head->n_next; /* 头指针指向下一个资源 */  
            /* 打印输出 */  
            printf("consume %d\n", p_node->n_number);  
  
            free(p_node); /* 拿到资源后释放节点占用的内存 */  
        }  
  
        pthread_mutex_unlock(&mutex); /* 临界区数据操作完毕，释放互斥锁 */  
        sem_post(&empty_sem); /* 发送一个空信号量给生产者 */  
    }  
}  
/* 生产者线程入口函数 */  
static void* product(void* parameter)  
{  
    int count = 0;  
    struct node *p_node;  
  
    while(1)  
    {  
        /* 动态分配一块结构体内存 */  
        p_node = (struct node*)malloc(sizeof(struct node));  
        if (p_node != NULL)  
        {  
            p_node->n_number = count++;  
            pthread_mutex_lock(&mutex); /* 需要操作 head 这个临界资源，先加锁 */  
  
            p_node->n_next = head;  
            head = p_node; /* 往链头插入数据 */  
  
            pthread_mutex_unlock(&mutex); /* 解锁 */  
            printf("produce %d\n", p_node->n_number);  
  
            sem_post(&full_sem); /* 发送一个满信号量给消费者 */  
        }  
        else  
        {  
            printf("product malloc node failed!\n");  
        }  
    }  
}
```

```

        break;
    }
    sleep(2); /* 休眠 2 秒 */
    sem_wait(&empty_sem); /* 等待消费者发送空信号量 */
}
}

int rt_application_init()
{
    int result;

    sem_init(&empty_sem,NULL,0);
    sem_init(&full_sem,NULL,0);
/* 创建生产者线程，属性为默认值，入口函数是 product，入口函数参数为 NULL*/
    result = pthread_create(&tid1,NULL,product,NULL);
    check_result("product thread created",result);

/* 创建消费者线程，属性为默认值，入口函数是 consumer，入口函数参数是 NULL */
    result = pthread_create(&tid2,NULL,consumer,NULL);
    check_result("consumer thread created",result);

    return 0;
}

```

29.8 消息队列

消息队列是另一种常用的线程间通讯方式，它能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。

消息队列主要操作包括：通过函数 `mq_open()` 创建或者打开，调用 `mq_send()` 发送一条消息到消息队列，调用 `mq_receive()` 从消息队列获取一条消息，当消息队列不在使用时，可以调用 `mq_unlink()` 删除消息队列。

POSIX 消息队列主要用于进程间通信，RT-Thread 操作系统的 POSIX 消息队列主要是基于 RT-Thread 内核消息队列的一个封装，主要还是用于系统内线程间的通讯。使用方式和 RT-Thread 内核的消息队列差不多。

29.8.1 消息队列控制块

每个消息队列对应一个消息队列控制块，创建消息队列前需要先定义一个消息队列控制块。消息队列控制块定义在 `mqueue.h` 头文件里。

```

struct mqdes
{
    rt_uint16_t refcount; /* 引用计数 */
    rt_uint16_t unlinked; /* 消息队列的分离状态，值为 1 表示消息队列已经分离 */
}

```

```
rt_mq_t mq;           /* RT-Thread 消息队列控制块 */
struct mqdes* next;   /* 指向下一个消息队列控制块 */
};

typedef struct mqdes* mqd_t; /* 消息队列控制块指针类型重定义 */
```

29.8.2 创建或打开消息队列

```
mqd_t mq_open(const char *name, int oflag, ...);
```

参数	描述
name	消息队列名称
oflag	消息队列打开方式
返回	—
消息队列句柄	成功
NULL	失败

此函数会根据消息队列的名字 name 创建一个新的消息队列或者打开一个已经存在的消息队列。Oflag 的可选值有 0、O_CREAT 或 O_CREAT|O_EXCL。如果 Oflag 设置为 O_CREAT 则会创建一个新的消息队列。如果 Oflag 设置 O_CREAT|O_EXCL，如果消息队列已经存在则会返回 NULL，如果不存在则会创建一个新的消息队列。如果 Oflag 设置为 0，消息队列不存在则会返回 NULL。

29.8.3 分离消息队列

```
int mq_unlink(const char *name);
```

参数	描述
name	消息队列名称
返回	—
0	成功
-1	失败

此函数会根据消息队列名称 name 查找消息队列，若找到，则将消息队列置为分离状态，之后若持有计数为 0，则删除消息队列，并释放消息队列占有的资源。

29.8.4 关闭消息队列

```
int mq_close(mqd_t mqdes);
```

参数	描述
mqdes	消息队列句柄
返回	—
0	成功
-1	失败

当一个线程终止时，会对其占用的消息队列执行此关闭操作。不论线程是自愿终止还是非自愿终止都会执行这个关闭操作，相当于是消息队列的持有计数减 1，若减 1 后持有计数为 0，且消息队列处于分离状态，则会删除 mqdes 消息队列并释放其占有的资源。

29.8.5 阻塞方式发送消息

```
int mq_send(mqd_t mqdes,
            const char *msg_ptr,
            size_t msg_len,
            unsigned msg_prio);
```

参数	描述
mqdes	消息队列句柄，不能为 NULL
sg_ptr	指向要发送的消息的指针，不能为 NULL
msg_len	发送的消息的长度
msg_prio	RT-Thread 未实现参数
返回	—
0	成功
-1	失败

此函数用来向 mqdes 消息队列发送一条消息，是 rt_mq_send() 函数的封装。此函数把 msg_ptr 指向的消息添加到 mqdes 消息队列中，发送的消息长度 msg_len 必须小于或者等于创建消息队列时设置的最大消息长度。

如果消息队列已经满，即消息队列中的消息数量等于最大消息数，发送消息的线程或者中断程序会收到一个错误码 (-RT_EFULL)。

29.8.6 指定阻塞时间发送消息

```
int mq_timedsend(mqd_t mqdes,
                  const char *msg_ptr,
                  size_t msg_len,
                  unsigned msg_prio,
```

```
const struct timespec *abs_timeout);
```

参数	描述
mqdes	消息队列句柄, 不能为 NULL
msg_ptr	指向要发送的消息的指针, 不能为 NULL
msg_len	发送的消息的长度
msg_prio	RT-Thread 未实现参数
abs_timeout	指定的等待时间, 单位是操作系统时钟节拍 (OS Tick)
返回	—
0	成功
-1	失败

目前 RT-Thread 不支持指定阻塞时间发送消息, 但是函数接口已经实现, 相当于调用 `mq_send()`。

29.8.7 阻塞方式接受消息

```
ssize_t mq_receive(mqd_t mqdes,
                   char *msg_ptr,
                   size_t msg_len,
                   unsigned *msg_prio);
```

参数	描述
mqdes	消息队列句柄, 不能为 NULL
msg_ptr	指向要发送的消息的指针, 不能为 NULL
msg_len	发送的消息的长度
msg_prio	RT-Thread 未实现参数
返回	—
消息长度	成功
-1	失败

此函数会把 `mqdes` 消息队列里面最老的消息移除消息队列, 并把消息放到 `msg_ptr` 指向的内存里。如果消息队列为空, 调用 `mq_receive()` 函数的线程将会阻塞, 直到消息队列中消息可用。

29.8.8 指定阻塞时间接受消息

```
ssize_t mq_timedreceive(mqd_t mqdes,
                        char *msg_ptr,
```

```
    size_t msg_len,
    unsigned *msg_prio,
    const struct timespec *abs_timeout);
```

参数	描述
mqdes	消息队列句柄, 不能为 NULL
msg_ptr	指向要发送的消息的指针, 不能为 NULL
msg_len	发送的消息的长度
msg_prio	RT-Thread 未实现参数
abs_timeout	指定的等待时间, 单位是操作系统时钟节拍 (OS Tick)
返回	—
消息长度	成功
-1	失败

此函数和 mq_receive() 函数的区别在于, 若消息队列为空, 线程将阻塞 abs_timeout 时长, 超时后函数直接返回 -1, 线程将被唤醒由阻塞态进入就绪态。

29.8.9 消息队列示例代码

这个程序会创建 3 个线程, 线程 2 从消息队列接受消息, 线程 2 和线程 3 往消息队列发送消息。

```
#include <mqueue.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;
static pthread_t tid3;
/* 消息队列句柄 */
static mqd_t mqueue;

/* 函数返回值检查函数 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}
```

```
/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    char buf[128];
    int result;

    while (1)
    {
        /* 从消息队列中接收消息 */
        result = mq_receive(mqueue, &buf[0], sizeof(buf), 0);
        if (result != -1)
        {
            /* 输出内容 */
            printf("thread1 recv [%s]\n", buf);
        }

        /* 休眠 1 秒 */
        // sleep(1);
    }
}

/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int i, result;
    char buf[] = "message2 No.x";

    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            buf[sizeof(buf) - 2] = '0' + i;

            printf("thread2 send [%s]\n", buf);
            /* 发送消息到消息队列中 */
            result = mq_send(mqueue, &buf[0], sizeof(buf), 0);
            if (result == -1)
            {
                /* 消息队列满， 延迟 1s 时间 */
                printf("thread2:message queue is full, delay 1s\n");
                sleep(1);
            }
        }

        /* 休眠 2 秒 */
        sleep(2);
    }
}

/* 线程 3 入口函数 */
static void* thread3_entry(void* parameter)
```

```
{  
    int i, result;  
    char buf[] = "message3 No.x";  
  
    while (1)  
    {  
        for (i = 0; i < 10; i++)  
        {  
            buf[sizeof(buf) - 2] = '0' + i;  
  
            printf("thread3 send [%s]\n", buf);  
            /* 发送消息到消息队列中 */  
            result = mq_send(mqueue, &buf[0], sizeof(buf), 0);  
            if (result == -1)  
            {  
                /* 消息队列满， 延迟 1s 时间 */  
                printf("thread3:message queue is full, delay 1s\n");  
                sleep(1);  
            }  
        }  
  
        /* 休眠 2 秒 */  
        sleep(2);  
    }  
}  
/* 用户应用入口 */  
int rt_application_init()  
{  
    int result;  
    struct mq_attr mqstat;  
    int oflag = O_CREAT|O_RDWR;  
#define MSG_SIZE      128  
#define MAX_MSG       128  
    memset(&mqstat, 0, sizeof(mqstat));  
    mqstat.mq_maxmsg = MAX_MSG;  
    mqstat.mq_msgsize = MSG_SIZE;  
    mqstat.mq_flags = 0;  
    mqueue = mq_open("mqueue1", O_CREAT, 0777, &mqstat);  
  
    /* 创建线程 1，线程入口是 thread1_entry，属性参数设为 NULL 选择默认值，入口参数  
     * 为 NULL*/  
    result = pthread_create(&tid1, NULL, thread1_entry, NULL);  
    check_result("thread1 created", result);  
  
    /* 创建线程 2，线程入口是 thread2_entry，属性参数设为 NULL 选择默认值，入口参数  
     * 为 NULL*/  
    result = pthread_create(&tid2, NULL, thread2_entry, NULL);  
    check_result("thread2 created", result);  
}
```

```

/* 创建线程 3, 线程入口是 thread3_entry, 属性参数设为 NULL 选择默认值, 入口参数
   为 NULL*/
result = pthread_create(&tid3,NULL,thread3_entry,NULL);
check_result("thread3 created",result);

return 0;
}

```

29.9 线程高级编程

本章节会对一些很少使用的属性对象及相关函数做详细介绍。

RT-Thread 实现的线程属性包括线程栈大小、线程优先级、线程分离状态、线程调度策略。pthread_create() 使用属性对象前必须先对属性对象进行初始化。设置线程属性之类的 API 函数应在创建线程之前就调用。线程属性的变更不会影响到已创建的线程。

线程属性结构 pthread_attr_t 定义在 pthread.h 头文件里。线程属性结构如下：

```

/* pthread_attr_t 类型重定义 */
typedef struct pthread_attr pthread_attr_t;
/* 线程属性结构体 */
struct pthread_attr
{
    void*      stack_base;          /* 线程栈的地址 */
    rt_uint32_t stack_size;         /* 线程栈大小 */
    rt_uint8_t  priority;           /* 线程优先级 */
    rt_uint8_t  detachstate;        /* 线程的分离状态 */
    rt_uint8_t  policy;             /* 线程调度策略 */
    rt_uint8_t  inheritsched;       /* 线程的继承性 */
};

```

29.9.0.1 线程属性初始化及去初始化

线程属性初始化及去初始化函数如下所示：

```

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

```

参数	描述
attr	指向线程属性的指针
返回	—
0	成功

使用 `pthread_attr_init()` 函数会使用默认值初始化线程属性结构体 `attr`，等同于调用线程初始化函数时将此参数设置为 `NULL`，使用前需要定义一个 `pthread_attr_t` 属性对象，此函数必须在 `pthread_create()` 函数之前调用。

`pthread_attr_destroy()` 函数对 `attr` 指向的属性去初始化，之后可以再次调用 `pthread_attr_init()` 函数对此属性对象重新初始化。

29.9.0.2 线程的分离状态

设置线程的分离状态 / 获取线程的分离状态如下所示，默认情况下线程是非分离状态。

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int state);
int pthread_attr_getdetachstate(pthread_attr_t const *attr, int *state);
```

参数	描述
<code>attr</code>	指向线程属性的指针
<code>state</code>	线程分离状态
返回	—
0	成功

线程分离状态属性值 `state` 可以是 `PTHREAD_CREATE_JOINABLE`(非分离)和 `PTHREAD_CREATE_DETACHED`(分离)。

线程的分离状态决定一个线程以什么样的方式来回收自己运行结束后占用的资源。线程的分离状态有 2 种：`joinable` 或者 `detached`。当线程创建后，应该调用 `pthread_join()` 或者 `pthread_detach()` 回收线程结束运行后占用的资源。如果线程的分离状态为 `joinable` 其他线程可以调用 `pthread_join()` 函数等待该线程结束并获取线程返回值，然后回收线程占用的资源。分离状态为 `detached` 的线程不能被其他的线程所 `join`，自己运行结束后，马上释放系统资源。

29.9.0.3 线程的调度策略

设置 获取线程调度策略函数如下所示：

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t const *attr, int *policy);
```

只实现了函数接口， 默认不同优先级基于优先级调度，同一优先级时间片轮询调度

29.9.0.4 线程的调度参数

设置线程的优先级 / 获取线程的优先级函数如下所示：

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               struct sched_param const *param);
int pthread_attr_getschedparam(pthread_attr_t const *attr,
```

```
struct sched_param *param);
```

参数	描述
attr	指向线程属性的指针
param	指向调度参数的指针
返回	—
0	成功

pthread_attr_setschedparam() 函数设置线程的优先级。使用 param 对线程属性优先级赋值。

参数 struct sched_param 定义在 sched.h 里，结构如下：

```
struct sched_param
{
    int sched_priority; /* 线程优先级 */
};
```

结构体 sched_param 的成员 sched_priority 控制线程的优先级值。

29.9.0.5 线程的堆栈大小

设置 / 获取线程的堆栈大小的函数如下所示：

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stack_size);
int pthread_attr_getstacksize(pthread_attr_t const *attr, size_t *stack_size);
```

参数	描述
attr	指向线程属性的指针
stack_size	线程堆栈大小
返回	—
0	成功

pthread_attr_setstacksize() 函数可以设置堆栈大小，单位是字节。在大多数系统中需要做栈空间地址对齐（例如 ARM 体系结构中需要向 4 字节地址对齐）。

29.9.0.6 线程堆栈大小和地址

设置 / 获取线程的堆栈地址和堆栈大小的函数如下所示：

```
int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stack_base,
                         size_t stack_size);
```

```
int pthread_attr_getstack(pthread_attr_t const *attr,
                          void**stack_base,
                          size_t *stack_size);
```

参数	描述
attr	指向线程属性的指针
stack_size	线程堆栈大小
stack_base	线程堆栈地址
返回	—
0	成功

29.9.0.7 线程属性相关函数

设置 / 获取线程的作用域的函数如下所示：

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t const *attr);
```

参数	描述
attr	指向线程属性的指针
scope	线程作用域
返回	—
0	scope 为 PTHREAD_SCOPE_SYSTEM
EOPNOTSUPP	scope 为 PTHREAD_SCOPE_PROCESS
EINVAL	scope 为 PTHREAD_SCOPE_SYSTEM

29.9.0.8 线程属性示例代码

这个程序会初始化 2 个线程，它们拥有共同的入口函数，但是它们的入口参数不相同。最先创建的线程会使用提供的 attr 线程属性，另外一个线程使用系统默认的属性。线程的优先级是很重要的一个参数，因此这个程序会修改第一个创建的线程的优先级为 8，而系统默认的优先级为 24。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <sched.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;
```

```
/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程入口函数 */
static void* thread_entry(void* parameter)
{
    int count = 0;
    int no = (int) parameter; /* 获得线程的入口参数 */

    while (1)
    {
        /* 打印输出线程计数值 */
        printf("thread%d count: %d\n", no, count ++);

        sleep(2); /* 休眠 2 秒 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    pthread_attr_t attr;      /* 线程属性 */
    struct sched_param prio; /* 线程优先级 */

    prio.sched_priority = 8; /* 优先级设置为 8 */
    pthread_attr_init(&attr); /* 先使用默认值初始化属性 */
    pthread_attr_setschedparam(&attr,&prio); /* 修改属性对应的优先级 */

    /* 创建线程 1, 属性为 attr, 入口函数是 thread_entry, 入口函数参数是 1 */
    result = pthread_create(&tid1,&attr,thread_entry,(void*)1);
    check_result("thread1 created",result);

    /* 创建线程 2, 属性为默认值, 入口函数是 thread_entry, 入口函数参数是 2 */
    result = pthread_create(&tid2,NULL,thread_entry,(void*)2);
    check_result("thread2 created",result);

    return 0;
}
```

29.9.1 线程取消

取消是一种让一个线程可以结束其它线程运行的机制。一个线程可以对另一个线程发送一个取消请求。依据设置的不同，目标线程可能会置之不理，可能会立即结束也可能会将它推迟到下一个取消点才结束。

29.9.1.1 发送取消请求

可使用如下函数发送取消请求：

```
int pthread_cancel(pthread_t thread);
```

参数	描述
thread	线程句柄
返回	—
0	成功

此函数发送取消请求给 **thread** 线程。Thread 线程是否会对取消请求做出回应以及什么时候做出回应依赖于线程取消的状态及类型。

29.9.1.2 设置取消状态

可使用如下函数设置取消请求：

```
int pthread_setcancelstate(int state, int *oldstate);
```

参数	描述
state	有两种值：PTHREAD_CANCEL_ENABLE：取消使能 PTHREAD_CANCEL_DISABLE：取消不使能（线程创建时的默认值）
oldstate	保存原来的取消状态
返回	—
0	成功
EINVAL	state 非 PTHREAD_CANCEL_ENABLE 或者 PTHREAD_CANCEL_DISABLE

此函数设置取消状态，由线程自己调用。取消使能的线程将会对取消请求做出反应，而取消没有使能的线程不会对取消请求做出反应。

29.9.1.3 设置取消类型

可使用如下函数设置取消类型，由线程自己调用：

```
int pthread_setcanceltype(int type, int *oldtype);
```

参数	描述
type	有 2 种值：PTHREAD_CANCEL_DEFERRED：线程收到取消请求后继续运行至下一个取消点再结束。（线程创建时的默认值） PTHREAD_CANCEL_ASYNCHRONOUS：线程立即结束。
oldtype	保存原来的取消类型
返回	—
0	成功
EINVAL	state 非 PTHREAD_CANCEL_DEFERRED 或者 PTHREAD_CANCEL_ASYNCHRONOUS

29.9.1.4 设置取消点

可使用如下函数设置取消点：

```
void pthread_testcancel(void);
```

此函数在线程调用的地方创建一个取消点。主要由不包含取消点的线程调用，可以回应取消请求。如果在取消状态处于禁用状态下调用 `pthread_testcancel()`，则该函数不起作用。

29.9.1.5 取消点

取消点也就是线程接受取消请求后会结束运行的地方，根据 POSIX 标准，`pthread_join()`、`pthread_testcancel()`、`pthread_cond_wait()`、`pthread_cond_timedwait()`、`sem_wait()`等会引起阻塞的系统调用都是取消点。

RT-Thread 包含的所有取消点如下：

- `mq_receive()`
- `mq_send()`
- `mq_timedreceive()`
- `mq_timedsend()`
- `msgrcv()`
- `msgsnd()`
- `msync()`
- `pthread_cond_timedwait()`
- `pthread_cond_wait()`
- `pthread_join()`

- pthread_testcancel()
- sem_timedwait()
- sem_wait()
- pthread_rwlock_rdlock()
- pthread_rwlock_timedrdlock()
- pthread_rwlock_timedwrlock()
- pthread_rwlock_wrlock()

29.9.1.6 线程取消示例代码

此程序会创建 2 个线程，线程 2 开始运行后马上休眠 8 秒，线程 1 设置了自己的取消状态和类型，之后在一个无限循环里打印运行计数信息。线程 2 唤醒后向线程 1 发送取消请求，线程 1 收到取消请求后马上结束运行。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

/* 线程控制块 */
static pthread_t tid1;
static pthread_t tid2;

/* 函数返回值检查 */
static void check_result(char* str,int result)
{
    if (0 == result)
    {
        printf("%s successfully!\n",str);
    }
    else
    {
        printf("%s failed! error code is %d\n",str,result);
    }
}

/* 线程 1 入口函数 */
static void* thread1_entry(void* parameter)
{
    int count = 0;
    /* 设置线程 1 的取消状态使能，取消类型为线程收到取消点后马上结束 */
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    while(1)
    {
        /* 打印线程计数值输出 */
        printf("Thread 1: count = %d\n", count);
        count++;
        sleep(1);
    }
}
```

```

        printf("thread1 run count: %d\n",count++);
        sleep(2);      /* 休眠 2 秒 */
    }
}

/* 线程 2 入口函数 */
static void* thread2_entry(void* parameter)
{
    int count = 0;
    sleep(8);
    /* 向线程 1 发送取消请求 */
    pthread_cancel(tid1);
    /* 阻塞等待线程 1 运行结束 */
    pthread_join(tid1,NULL);
    printf("thread1 exited!\n");
    /* 线程 2 打印信息开始输出 */
    while(1)
    {
        /* 打印线程计数值输出 */
        printf("thread2 run count: %d\n",count++);
        sleep(2);      /* 休眠 2 秒 */
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    int result;
    /* 创建线程 1, 属性为默认值, 分离状态为默认值 joinable,
       入口函数是 thread1_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid1,NULL,thread1_entry,NULL);
    check_result("thread1 created",result);

    /* 创建线程 2, 属性为默认值, 分离状态为默认值 joinable,
       入口函数是 thread2_entry, 入口函数参数为 NULL */
    result = pthread_create(&tid2,NULL,thread2_entry,NULL);
    check_result("thread2 created",result);

    return 0;
}

```

29.9.2 一次性初始化

可使用如下函数一次性初始化：

```
int pthread_once(pthread_once_t * once_control, void (*init_routine) (void));
```

参数	描述
once_control	控制变量

参数	描述
init_routine	执行函数
返回	—
0	成功

有时候我们需要对一些变量只进行一次初始化。如果我们进行多次初始化程序就会出现错误。在传统的顺序编程中，一次性初始化经常通过使用布尔变量来管理。控制变量被静态初始化为 0，而任何依赖于初始化的代码都能测试该变量。如果变量值仍然为 0，则它能实行初始化，然后将变量置为 1。以后检查的代码将跳过初始化。

29.9.3 线程结束后清理

线程清理函数接口如下所示：

```
void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

参数	描述
execute	0 或 1，决定是否执行 cleanup 函数
routine	指向清理函数的指针
arg	传递给清理函数的参数

`pthread_cleanup_push()` 把指定的清理函数 `routine` 放到线程的清理函数链表里, `pthread_cleanup_pop()` 从清理函数链表头部取出第一项函数，若 `execute` 为非 0 值，则执行此函数。

29.9.4 其他线程相关函数

29.9.4.1 判断 2 个线程是否相等

```
int pthread_equal (pthread_t t1, pthread_t t2);
```

参数	描述
pthread_t	线程句柄
返回	—
0	不相等
1	相等

29.9.4.2 获取线程句柄

```
pthread_t pthread_self (void);
```

`pthread_self()` 返回调用线程的句柄。

29.9.4.3 获取最大最小优先级

```
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

参数	描述
policy	2 个值可选: SCHED_FIFO, SCHED_RR

`sched_get_priority_min()` 返回值为 0, RT-Thread 里为最大优先级, `sched_get_priority_max()` 返回值最小优先级。

29.9.5 互斥锁属性

RT-Thread 实现的互斥锁属性包括互斥锁类型和互斥锁作用域。

29.9.5.1 互斥锁属性初始化及去初始化

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

参数	描述
attr	指向互斥锁属性对象的指针
返回	—
0	成功
EINVAL	参数无效

`pthread_mutexattr_init()` 函数将使用默认值初始化 `attr` 指向的属性对象, 等同于调用 `pthread_mutex_init()` 函数时将属性参数设置为 NULL。

`pthread_mutexattr_destroy()` 函数将会对 `attr` 指向的属性对象去初始化, 之后可以调用 `pthread_mutexattr_init()` 函数重新初始化。

29.9.5.2 互斥锁作用域

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
int pthread_mutexattr_getpshared(pthread_mutexattr_t *attr, int *pshared);
```

参数	描述
type	互斥锁类型
pshared	有 2 个可选值: PTHREAD_PROCESS_PRIVATE: 默认值, 用于仅同步该进程中的线程。PTHREAD_PROCESS_SHARED: 用于同步该进程和其他进程中的线程。
返回	—
0	成功
EINVAL	参数无效

29.9.5.3 互斥锁类型

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type);
```

参数	描述
type	互斥锁类型
attr	指向互斥锁属性对象的指针
返回	—
0	成功
EINVAL	参数无效

互斥锁的类型决定了一个线程在获取一个互斥锁时的表现方式, RT-Thread 实现了 3 种互斥锁类型:

- PTHREAD_MUTEX_NORMAL: 普通锁, 当一个线程加锁以后, 其余请求锁的线程将形成一个等待队列, 并在解锁后按先进先出方式获得锁。如果一个线程在不首先解除互斥锁的情况下尝试重新获得该互斥锁, 不会产生死锁, 而是返回错误码, 和检错锁一样。
- PTHREAD_MUTEX_RECURSIVE: 嵌套锁, 允许一个线程对同一个锁成功获得多次, 需要相同次数的解锁释放该互斥锁。
- PTHREAD_MUTEX_ERRORCHECK: 检错锁, 如果一个线程在不首先解除互斥锁的情况下尝试重新获得该互斥锁, 则返回错误。这样就保证当不允许多次加锁时不会出现死锁。

29.9.6 条件变量属性

使用默认值 PTHREAD_PROCESS_PRIVATE 初始化条件变量属性 attr 可使用如下函数:

```
int pthread_condattr_init(pthread_condattr_t *attr);
```

参数	描述
attr	指向条件变量属性对象的指针
返回	—
0	成功
EINVAL	参数无效

29.9.6.1 获取条件变量作用域

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t *attr, int *pshared);
```

参数	描述
attr	指向条件变量属性对象的指针
返回	—
0	成功
EINVAL	参数无效

29.9.7 读写锁属性

29.9.7.1 初始化属性

```
int pthread_rwlockattr_init (pthread_rwlockattr_t *attr);
```

参数	描述
attr	指向读写锁属性的指针
返回	—
0	成功
-1	参数无效

该函数会使用默认值 PTHREAD_PROCESS_PRIVATE 初始化读写锁属性 attr。

29.9.7.2 获取作用域

```
int pthread_rwlockattr_getpshared (const pthread_rwlockattr_t *attr, int *pshared);
```

参数	描述
attr	指向读写锁属性的指针
pshared	指向保存读写锁作用域的指针
返回	—
0	成功
-1	参数无效

pshared 指向的内存保存的值为 PTHREAD_PROCESS_PRIVATE。

29.9.8 屏障属性

29.9.8.1 初始化属性

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

参数	描述
attr	指向屏障属性的指针
返回	—
0	成功
-1	参数无效

该函数会使用默认值 PTHREAD_PROCESS_PRIVATE 初始化屏障属性 attr。

29.9.8.2 获取作用域

```
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr, int *pshared);
```

参数	描述
attr	指向屏障属性的指针
pshared	指向保存屏障作用域数据的指针
返回	—
0	成功
-1	参数无效

29.9.9 消息队列属性

消息队列属性控制块如下：

```
struct mq_attr
{
    long mq_flags;          /* 消息队列的标志，用来表示是否阻塞 */
    long mq_maxmsg;         /* 消息队列最大消息数 */
    long mq_msgsize;        /* 消息队列每个消息的最大字节数 */
    long mq_curmsgs;        /* 消息队列当前消息数 */
};
```

29.9.1 获取属性

```
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

参数	描述
mqdes	指向消息队列控制块的指针
mqstat	指向保存获取数据的指针
返回	—
0	成功
-1	参数无效

第 30 章

电源管理组件

嵌入式系统低功耗管理的目的在于满足用户对性能需求的前提下，尽可能降低系统能耗以延长设备待机时间。高性能与有限的电池能量在嵌入式系统中矛盾最为突出，硬件低功耗设计与软件低功耗管理的联合应用成为解决矛盾的有效手段。现在的各种 MCU 都或多或少的在低功耗方面提供了管理接口。比如对主控时钟频率的调整、工作电压的改变、总线频率的调整甚至关闭、外围设备工作时钟的关闭等。有了硬件上的支持，合理的软件设计就成为节能的关键，一般可以把低功耗管理分为三个类别：

- 处理器电源管理主要实现方式：对 CPU 频率的动态管理，以及系统空闲时对工作模式的调整。
- 设备电源管理主要实现方式：关闭个别闲置设备
- 系统平台电源管理主要实现方式：针对特定系统平台的非常见设备具体定制。

随着物联网 (IoT) 的兴起，产品对功耗的需求越来越强烈。作为数据采集的传感器节点通常需要在电池供电时长期工作，而作为联网的 SOC 也需要有快速的响应功能和较低的功耗。

在产品开发的起始阶段，首先考虑是尽快完成产品的功能开发。在产品功能逐步完善之后，就需要加入电源管理 (Power Management，以下简称 PM) 功能。为了适应 IoT 的这种需求，RT-Thread 提供了电源管理组件。电源管理组件的理念是尽量透明，使得产品加入低功耗功能更加轻松。

30.1 PM 组件介绍

RT-Thread 的 PM 组件采用分层设计思想，分离架构和芯片相关的部分，提取公共部分作为核心。在对上层提供通用的接口同时，也让底层驱动对组件的适配变得更加简单。

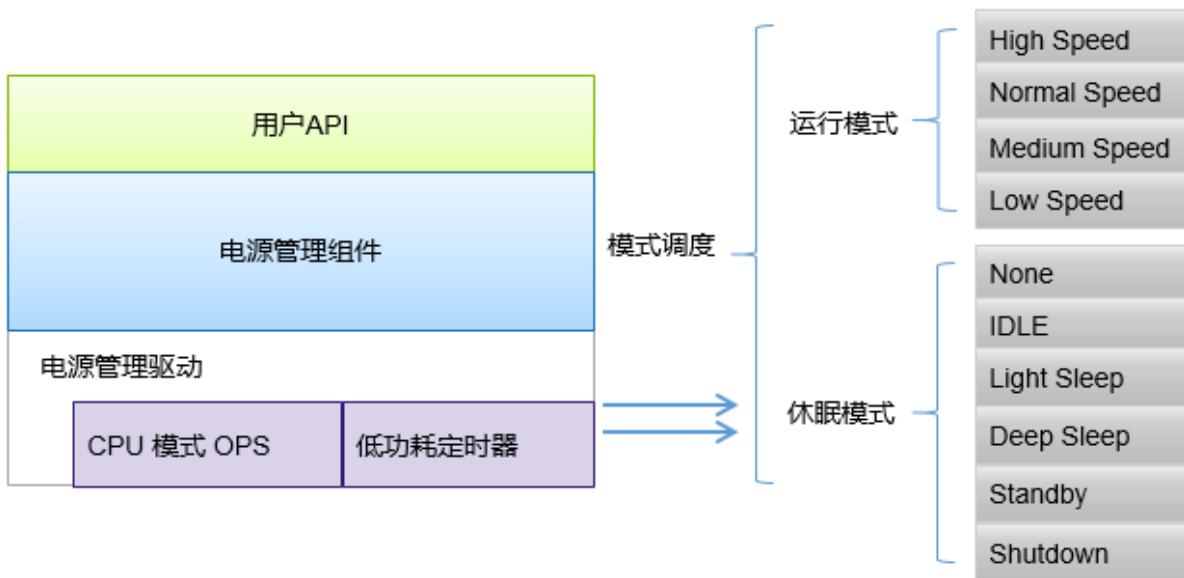


图 30.1: PM 组件概略图

30.1.1 主要特点

RT-Thread PM 组件主要特点如下所示：

- 基于模式来管理功耗，空闲时动态调整工作模式，支持多个等级的休眠。
- 对应用透明，组件在底层自动完成电源管理。
- 支持运行模式下动态变频，根据模式自动更新设备的频率配置，确保在不同的运行模式都可以正常工作。
- 支持设备电源管理，根据模式自动管理设备的挂起和恢复，确保在不同的休眠模式下可以正确的挂起和恢复。
- 支持可选的休眠时间补偿，让依赖 OS Tick 的应用可以透明使用。
- 向上层提供设备接口，如果打开了 devfs 组件，那么也可以通过文件系统接口访问。

30.1.2 工作原理

低功耗的本质是系统空闲时 CPU 停止工作，中断或事件唤醒后继续工作。在 RTOS 中，通常包含一个 IDLE 任务，该任务的优先级最低且一直保持就绪状态，当高优先级任务未就绪时，OS 执行 IDLE 任务。一般地，未进行低功耗处理时，CPU 在 IDLE 任务中循环执行空指令。RT-Thread 的电源管理组件在 IDLE 任务中，通过对 CPU、时钟和设备等进行管理，从而有效降低系统的功耗。

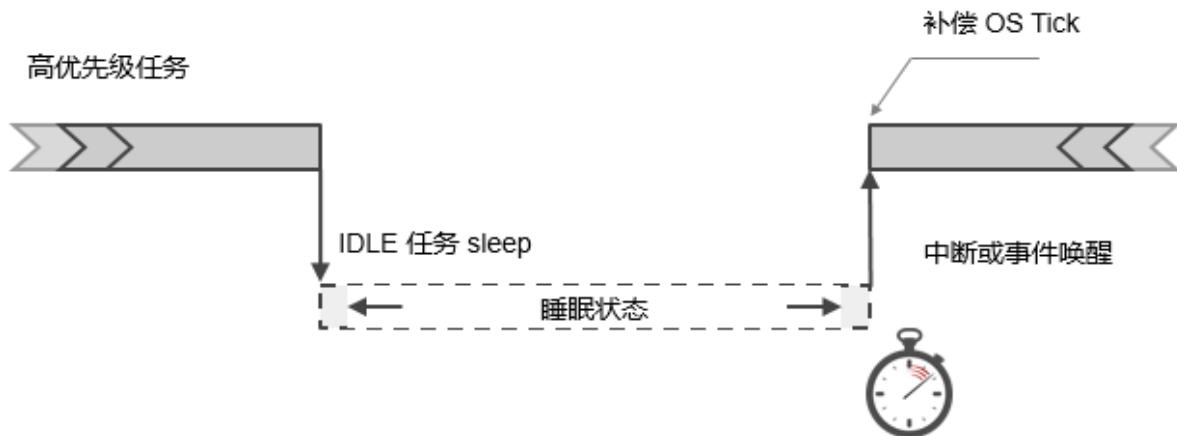


图 30.2: PM 工作原理

在上图所示，当高优先级任务运行结束或被挂起时，系统将进入 IDLE 任务中。在 IDLE 任务执行后，它将判断系统是否可以进入到休眠状态（以节省功耗）。如果可以进入休眠，将根据芯片情况关闭部分硬件模块，OS Tick 也非常有可能进入暂停状态。此时电源管理框架会根据系统定时器情况，计算出下一个超时时间点，并设置低功耗定时器，让设备能够在这个时刻点唤醒，并进行后续的工作。当系统被（低功耗定时器中断或其他唤醒中断源）唤醒后，系统也需要知道睡眠时间长度是多少，并对 OS Tick 进行补偿，让系统的 OS tick 值调整为一个正确的值。

30.2 设计架构

在 RT-Thread PM 组件中，外设或应用通过投票机制对所需的功耗模式进行投票，当系统空闲时，根据投票数决策出合适的功耗模式，调用抽象接口，控制芯片进入低功耗状态，从而降低系统功耗。当未进行任何投票时，会以默认模式进入（通常为空闲模式）。与应用不同，某些外设可能在进入低功耗状态时执行特定操作，退出低功耗时采取措施恢复，此时可以通过注册 PM 设备来实现。通过注册 PM 设备，在进入低功耗状态之前，会触发注册设备的 `suspend` 回调，开发者可在回调里执行自己的操作；类似地，从低功耗状态退出时，也会触发 `resume` 回调。

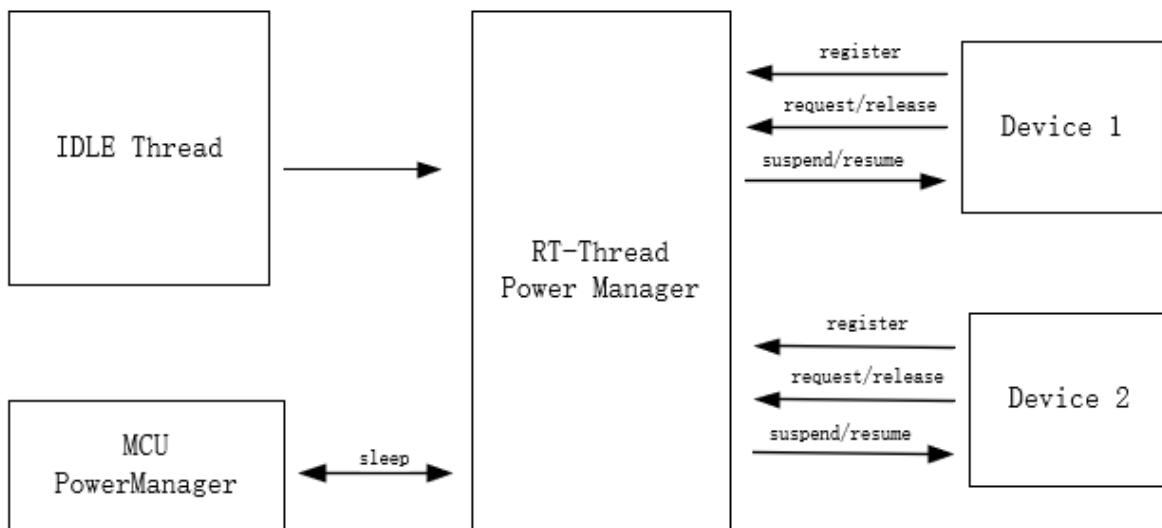


图 30.3: PM 设计架构

30.3 低功耗状态和模式

RT-Thread PM 组件将系统划分为两种状态: 运行状态 (RUN) 和休眠状态 (Sleep)。运行状态控制 CPU 的频率, 适用于变频场景; 休眠状态根据 SOC 特性实现休眠 CPU, 以降低功耗。两种状态分别使用不同的 API 接口, 独立控制。

- 休眠状态

休眠状态也就是通常意义上的低功耗状态, 通过关闭外设、执行 SOC 电源管理接口, 降低系统功耗。休眠状态又分为六个模式, 呈现为金字塔的形式。随着模式增加, 功耗逐级递减的特点。下面是休眠状态下模式的定义, 开发者可根据具体的 SOC 实现相应的模式, 但需要遵循功耗逐级降低的特点。

模式	级别	描述
PM_SLEEP_MODE_NONE	0	系统处于活跃状态, 未采取任何的降低功耗状态
PM_SLEEP_MODE_IDLE	1	空闲模式, 该模式在系统空闲时停止 CPU 和部分时钟, 任意事件或中断均可以唤醒
PM_SLEEP_MODE_LIGHT	2	轻度睡眠模式, CPU 停止, 多数时钟和外设停止, 唤醒后需要进行时间补偿
PM_SLEEP_MODE_DEEP	3	深度睡眠模式, CPU 停止, 仅少数低功耗外设工作, 可被特殊中断唤醒
PM_SLEEP_MODE_STANDBY	4	待机模式, CPU 停止, 设备上下文丢失(可保存至特殊外设), 唤醒后通常复位

模式	级别	描述
PM_SLEEP_MODE_SHUTDOWN 5		关断模式，比 Standby 模式功耗更低，上下文通常不可恢复，唤醒后复位

注意: 因各家芯片差异，功耗管理的实现也不尽相同，上述的描述仅给出一些推荐的场景，并非一定要实现所有。开发者可根据自身情况选择其中的几种进行实现，但是需要遵循级别越高，功耗越低的原则！

- 运行状态

运行状态通常用于改变 CPU 的运行频率，独立于休眠模式。当前运行状态划分了四个等级：高速、正常、中速、低速，如下：

模式	描述
PM_RUN_MODE_HIGH_SPEED	高速模式，适用于一些超频的场景
PM_RUN_MODE_NORMAL_SPEED	正常模式，该模式作为默认的运行状态
PM_RUN_MODE_MEDIUM_SPEED	中速模式，降低 CPU 运行速度，从而降低运行功耗
PM_RUN_MODE_LOW_SPEED	低速模式，CPU 频率进一步降低

30.3.1 模式的请求和释放

在 PM 组件里，上层应用可以通过请求和释放休眠模式主动参与功耗管理。应用可以根据场景请求不同的休眠模式，并在处理完毕后释放，只要有任意一个应用或设备请求高等级的功耗模式，就不会切换到比它更低的模式。因此，休眠模式的请求和释放的操作通常成对出现，可用于对某个阶段进行保护，如外设的 DMA 传输过程。

30.3.2 对模式变化敏感的设备

在 PM 组件里，切换到新的运行模式可能会导致 CPU 频率发生变化，如果外设和 CPU 共用一部分时钟，那外设的时钟就会受到影响；在进入新的休眠模式，大部分时钟源会被停止，如果外设不支持休眠的冻结功能，那么从休眠唤醒的时候，外设的时钟就需要重新配置外设。所以 PM 组件里支持了 PM 模式敏感的 PM 设备。使得设备在切换到新的运行模式或者新的休眠模式都能正常的工作。该功能需要底层驱动实现相关的接口并注册为对模式变化敏感的设备。

30.4 调用流程

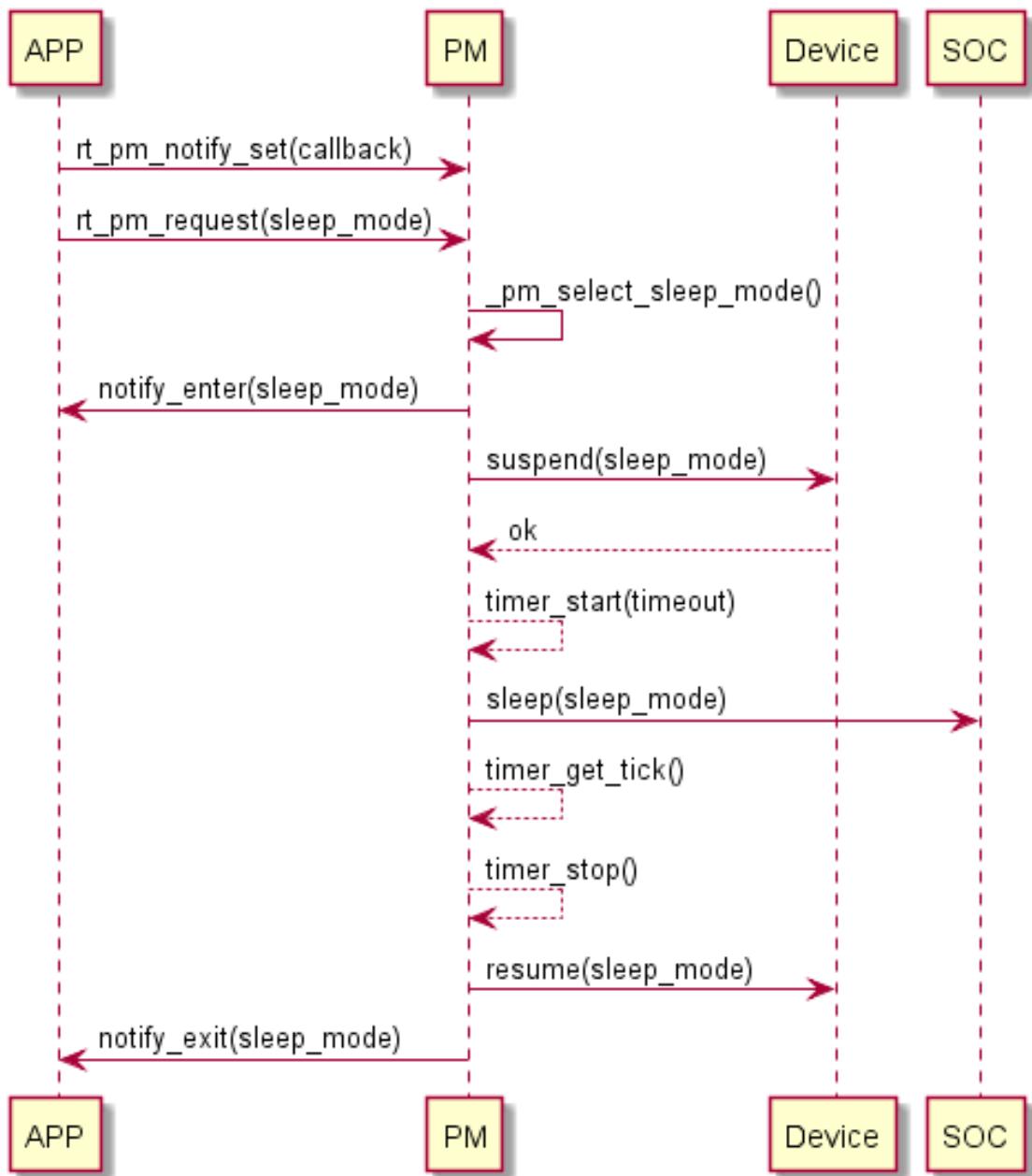


图 30.4: PM 时序图

首先应用设置进出休眠状态的回调函数，然后调用 `rt_pm_request` 请求休眠模式，触发休眠操作；PM 组件在系统空闲时检查休眠模式计数，根据投票数给出推荐的模式；接着 PM 组件调用 `notify` 通知应用，告知即将进入休眠模式；然后对注册的 PM 设备执行挂起操作，返回 OK 后执行 SOC 实现的的休眠模式，系统进入休眠状态（如果使能时间补偿，休眠之前会先启动低功耗定时器）。此时 CPU 停止工作，等待事件或者中断唤醒。当系统被唤醒后，由于全局中断为关闭状态，系统继续从该处执行，获取睡眠时间补偿系统的心跳，依次唤醒设备，通知应用从休眠模式退出。如此一个周期执行完毕，退出，等待系统下次空闲。

30.5 API 介绍

- 请求休眠模式

```
void rt_pm_request(uint8_t sleep_mode);
```

参数	模式
sleep_mode	请求的休眠模式等级

sleep_mode 取以下枚举值：

```
enum
{
    /* sleep modes */
    PM_SLEEP_MODE_NONE = 0,      /* 活跃状态 */
    PM_SLEEP_MODE_IDLE,         /* 空闲模式（默认） */
    PM_SLEEP_MODE_LIGHT,        /* 轻度睡眠模式 */
    PM_SLEEP_MODE_DEEP,         /* 深度睡眠模式 */
    PM_SLEEP_MODE_STANDBY,      /* 待机模式 */
    PM_SLEEP_MODE_SHUTDOWN,     /* 关断模式 */
    PM_SLEEP_MODE_MAX,
};
```

调用该函数会将对应的模式计数加 1，并锁住该模式。此时如果请求更低级别的功耗模式，将无法进入，只有释放（解锁）先前请求的模式后，系统才能进入更低的模式；向更高的功耗模式请求则不受此影响。该函数需要和 `rt_pm_release` 配合使用，用于对某一阶段或过程进行保护。

- 释放休眠模式

```
void rt_pm_release(uint8_t sleep_mode);
```

参数	模式
sleep_mode	释放的休眠模式等级

调用该函数会将对应的模式计数减 1，配合 `rt_pm_request` 使用，释放先前请求的模式。

- 设置运行模式

```
int rt_pm_run_enter(uint8_t run_mode);
```

参数	模式
run_mode	设置的运行模式等级

run_mode 可以取以下枚举值：

```
enum
{
    /* run modes */
    PM_RUN_MODE_HIGH_SPEED = 0,      /* 高速 */
    PM_RUN_MODE_NORMAL_SPEED,        /* 正常（默认） */
    PM_RUN_MODE_MEDIUM_SPEED,        /* 中速 */
    PM_RUN_MODE_LOW_SPEED,           /* 低速 */
    PM_RUN_MODE_MAX,
};
```

调用该函数改变 CPU 的运行频率，从而降低运行时的功耗。此函数只提供级别，具体的 CPU 频率应在移植阶段视实际情况而定。

- 设置进入/退出休眠模式的回调通知

```
void rt_pm_notify_set(void (*notify)(uint8_t event, uint8_t mode, void *data), void
                      *data);
```

参数	模式
notify	应用的回调函数
data	私有数据

event 为以下两个枚举值，分别标识进入/退出休眠模式。

```
enum
{
    RT_PM_ENTER_SLEEP = 0,      /* 进入休眠模式 */
    RT_PM_EXIT_SLEEP,          /* 退出休眠模式 */
};
```

30.6 使用说明

- 设置低功耗等级

如果系统需要进入指定指定等级的低功耗，可通过调用 `rt_pm_request` 实现。如进入深度睡眠模式：

```
/* 请求进入深度睡眠模式 */
```

```
rt_pm_request(PM_SLEEP_MODE_DEEP);
```

注意: 如果程序的其他地方请求了更高的功耗模式, 如 Light Mode 或者 Idle Mode, 则需要释放相应的模式后, 深度睡眠模式才能被进入。

- 保护某个阶段或者过程

特殊情况下, 比如某个阶段并不允许系统进入更低的功耗模式, 此时可以通过 `rt_pm_request` 和 `rt_pm_release` 对该过程进行保护。如 I2C 读取数据期间, 不允许进入深度睡眠模式(可能会导致外设停止工作), 因此可以做如下处理:

```
/* 请求轻度睡眠模式(I2C外设该模式下正常工作) */
rt_pm_request(PM_SLEEP_MODE_LIGHT);

/* 读取数据过程 */

/* 释放该模式 */
rt_pm_release(PM_SLEEP_MODE_LIGHT);
```

- 改变 CPU 运行频率

降低运行频率能有效减少系统的功耗, 通过 `rt_pm_run_enter` 接口改变 CPU 的运行频率。一般地, 降频率意味着 CPU 性能降低、处理速度减慢, 可能会导致任务的执行时间增长, 需要合理进行权衡。

```
/* 进入中等速度运行模式 */
rt_pm_run_enter(PM_RUN_MODE_MEDIUM_SPEED);
```

30.7 移植说明

低功耗管理是一项十分细致的任务, 开发者在移植时, 不仅需要充分了解芯片本身的功能管理, 还需熟悉板卡的外围电路, 进入低功耗状态时逐一处理, 避免出现外围电路漏电拉升整体功耗的情况。RT-Thread PM 组件对各部分进行抽象, 提供不同的 ops 接口供开发者适配。移植时需要关注的部分如下:

```
/**
 * low power mode operations
 */
struct rt_pm_ops
{
    /* sleep 接口用于适配芯片相关的低功耗特性 */
    void (*sleep)(struct rt_pm *pm, uint8_t mode);
    /* run 接口用于运行模式的变频和变电压 */
    void (*run)(struct rt_pm *pm, uint8_t mode);
    /* 以下三个接口用于心跳停止后启动低功耗定时器以补偿心跳 */
    void (*timer_start)(struct rt_pm *pm, rt_uint32_t timeout);
    void (*timer_stop)(struct rt_pm *pm);
    rt_tick_t (*timer_get_tick)(struct rt_pm *pm);
```

```

};

/* 该 ops 用于管理外设的功耗 */
struct rt_device_pm_ops
{
    /* 进入休眠模式之前挂起外设，返回非 0 表示外设未就绪，不能进入 */
    int (*suspend)(const struct rt_device *device, uint8_t mode);
    /* 从休眠模式退出后恢复外设 */
    void (*resume)(const struct rt_device *device, uint8_t mode);
    /* 运行状态下模式改变通知外设处理 */
    int (*frequency_change)(const struct rt_device *device, uint8_t mode);
};

/* 注册一个 PM 设备 */
void rt_pm_device_register(struct rt_device *device, const struct rt_device_pm_ops *ops);

```

- 芯片自身的功耗特性

```
void (*sleep)(struct rt_pm *pm, uint8_t mode);
```

各个芯片对低功耗模式的定义和管理不同，PM 组件将芯片相关的特性抽象为 sleep 接口。该接口适配芯片相关的低功耗管理，当要进入不同的休眠模式时，一些硬件相关的配置，保存等相关处理。

- 休眠的时间补偿

```

void (*timer_start)(struct rt_pm *pm, rt_uint32_t timeout);
void (*timer_stop)(struct rt_pm *pm);
rt_tick_t (*timer_get_tick)(struct rt_pm *pm);

```

在某些休眠模式下 (Light Sleep 或 Deep Sleep)，内核心跳定时器可能会被停止，此时需要对启动一个定时器对休眠的时间进行计量，唤醒后对心跳补偿。时间补偿的定时器必须在该模式下仍能够正常工作，并唤醒系统，否则没有意义！

`timer_start`：启动低功耗定时器，入参为最近的下次任务就绪时间；`timer_get_tick`：获取系统被唤醒的睡眠时间；`timer_stop`：用于系统唤醒后停止低功耗定时器。

注意：休眠模式的时间补偿需要在初始化阶段通过设置 `timer_mask` 的对应模式的 bit 控制开启。例如需要开启 Deep Sleep 模式下的时间补偿，在实现 timer 相关的 ops 接口后，初始化时设置相应的 bit：

```

rt_uint8_t timer_mask = 0;

/* 设置 Deep Sleep 模式对应的 bit，使能休眠时间补偿 */
timer_mask = 1UL << PM_SLEEP_MODE_DEEP;

/* initialize system pm module */
rt_system_pm_init(&_ops, timer_mask, RT_NULL);

```

- 运行模式变频

```
void (*run)(struct rt_pm *pm, uint8_t mode);
```

运行模式的变频通过适配 `rt_pm_ops` 中的 `run` 接口实现，根据使用场景选择合适的频率。

- 外设的功耗管理

外设的功耗处理是低功耗管理系统的一个重要部分，在进入某些级别的休眠模式时，通常需要对一些外设进行处理，如清空 DMA，关闭时钟或是设置 IO 为复位状态；并在退出休眠后进行恢复。此类情况可以通过 `rt_pm_device_register` 接口注册 PM 设备，在进入/退出休眠模式时，会执行注册设备的 `suspend` 和 `resume` 回调；运行模式下的频率改变同样会触发设备的 `frequency_change` 回调。

更详细的移植案例可以参考 RT-Thread 仓库中的 `stm32l476-nucleo` bsp。

30.8 MSH 命令

30.8.1 请求休眠模式

可以使用 `pm_request` 命令请求模式，使用示例如下所示：

```
msh />pm_request 0
msh />
```

参数取值为 0-5，分别对应以下枚举值：

```
enum
{
    /* sleep modes */
    PM_SLEEP_MODE_NONE = 0,      /* 活跃状态 */
    PM_SLEEP_MODE_IDLE,         /* 空闲模式（默认） */
    PM_SLEEP_MODE_LIGHT,        /* 轻度睡眠模式 */
    PM_SLEEP_MODE_DEEP,         /* 深度睡眠模式 */
    PM_SLEEP_MODE_STANDBY,      /* 待机模式 */
    PM_SLEEP_MODE_SHUTDOWN,     /* 关断模式 */
    PM_SLEEP_MODE_MAX,
};
```

30.8.2 释放休眠模式

可以使用 `pm_release` 命令释放模式，参数取值为 0-5，使用示例如下所示：

```
msh />pm_release 0
msh />
```

30.8.3 设置运行模式

可以使用 `pm_run` 命令切换运行模式，参数取值 0-3，使用示例如下所示

```
msh />pm_run 2
msh />
```

参数取值 0-3

```
enum
{
    /* run modes */
    PM_RUN_MODE_HIGH_SPEED = 0,
    PM_RUN_MODE_NORMAL_SPEED,
    PM_RUN_MODE_MEDIUM_SPEED,
    PM_RUN_MODE_LOW_SPEED,
    PM_RUN_MODE_MAX,
};
```

30.8.4 查看模式状态

可以使用 `pm_dump` 命令查看 PM 组件的模式状态，使用示例如下所示：

```
msh >
msh >pm_dump
| Power Management Mode | Counter | Timer |
+-----+-----+-----+
|      None Mode |      0 |      0 |
|      Idle Mode |      0 |      0 |
| LightSleep Mode |      1 |      0 |
| DeepSleep Mode |      0 |      1 |
|   Standby Mode |      0 |      0 |
| Shutdown Mode |      0 |      0 |
+-----+-----+-----+
pm current sleep mode: LightSleep Mode
pm current run mode: Normal Speed
msh >
```

在 `pm_dump` 的模式列表里，休眠模式的优先级是从高到低排列，`Counter` 一栏标识请求的计数值，图中表明 `LightSleep` 模式被请求一次，因此当前工作在轻度休眠状态；`Timer` 一栏标识是否开启睡眠时间补偿，图中仅深度睡眠 (`DeepSleep`) 模式进行时间补偿。最下面分别标识当前所处的休眠模式及运行模式等级。

30.9 常见问题及调试方法

- 系统进入低功耗模式后功耗偏高

根据外围电路，检查设备是否处于合理状态，避免出现外设漏电的情况；根据产品自身情况，关闭相应休眠模式期间不使用的外设和时钟。

- 无法进入更低等级的功耗

检查是否未释放高等级的功耗模式，RT-Thread 的 PM 组件使用 `rt_pm_request` 请求休眠模式，当请求高功耗模式后，未进行释放，系统将无法切换至更低等级的功耗。例如，在请求 Light Sleep 模式后，接着请求 Deep Sleep 模式，此时系统仍然处于 Light Sleep 模式。通过调用接口 `rt_pm_request` 释放 Light Sleep 模式，系统会自动切换到 Deep Sleep 模式。

第 31 章

SCons 构建工具

31.1 SCons 简介

SCons 是一套由 Python 语言编写的开源构建系统，类似于 GNU Make。它采用不同于通常 `Makefile` 文件的方式，而是使用 `SConstruct` 和 `SConscript` 文件来替代。这些文件也是 Python 脚本，能够使用标准的 Python 语法来编写。所以在 `SConstruct`、`SConscript` 文件中可以调用 Python 标准库进行各类复杂的处理，而不局限于 `Makefile` 设定的规则。

在 [SCons](#) 的网站上可以找到详细的 SCons 用户手册，本章节讲述 SCons 的基本用法，以及如何在 RT-Thread 中用好 SCons 工具。

31.1.1 什么是构建工具

构建工具 (software construction tool) 是一种软件，它可以根据一定的规则或指令，将源代码编译成可执行的二进制程序。这是构建工具最基本也是最重要的功能。实际上构建工具的功能不止于此，通常这些规则有一定的语法，并组织成文件。这些文件用来控制构建工具的行为，在完成软件构建之外，也可以做其他事情。

目前最流行的构建工具是 GNU Make。很多知名开源软件，如 Linux 内核就采用 Make 构建。Make 通过读取 `Makefile` 文件来检测文件的组织结构和依赖关系，并完成 `Makefile` 中所指定的命令。

由于历史原因，`Makefile` 的语法比较混乱，不利于初学者学习。此外在 Windows 平台上使用 Make 也不方便，需要安装 Cygwin 环境。为了克服 Make 的种种缺点，人们开发了其他构建工具，如 CMake 和 SCons 等。

31.1.2 RT-Thread 构建工具

RT-Thread 早期使用 Make/Makefile 构建。从 0.3.x 开始，RT-Thread 开发团队逐渐引入了 SCons 构建系统，引入 SCons 唯一的目是：使大家从复杂的 `Makefile` 配置、IDE 配置中脱离出来，把精力集中在 RT-Thread 功能开发上。

有些读者可能会有些疑惑，这里介绍的构建工具与 IDE 有什么不同呢？IDE 通过图形化界面的操作来完成构建。大部分 IDE 会根据用户所添加的源码生成类似 `Makefile` 或 `SConscript` 的脚本文件，在底层调用类似 Make 或 SCons 的工具来构建源码。

31.1.3 安装 SCons

在使用 SCons 系统前需要在 PC 主机中安装它，因为它是 Python 语言编写的，所以在使用 SCons 之前需要安装 Python 运行环境。

RT-Thread 提供的 Env 配置工具带有 SCons 和 Python，因此在 windows 平台使用 SCons 则不需要安装这两个软件。

在 Linux、BSD 环境中 Python 应该已经默认安装了，一般也是 2.x 版本系列的 Python 环境。这时只需要安装 SCons 即可，例如在 Ubuntu 中可以使用如下命令安装 SCons：

```
sudo apt-get install scons
```

31.2 SCons 基本功能

RT-Thread 构建系统支持多种编译器。目前支持的编译器包括 ARM GCC、MDK、IAR、VisualStudio、Visual DSP。主流的 ARM Cortex M0、M3、M4 平台，基本上 ARM GCC、MDK、IAR 都是支持的。有一些 BSP 可能仅支持一种，读者可以阅读该 BSP 目录下的 rtconfig.py 里的 CROSS_TOOL 选项查看当前支持的编译器。

如果是 ARM 平台的芯片，则可以使用 Env 工具，输入 scons 命令直接编译 BSP，这时候默认使用的是 ARM GCC 编译器，因为 Env 工具带有 ARM GCC 编译器。如下图所示使用 scons 命令编译 stm32f10x-HAL BSP，后文讲解 SCons 也将基于这个 BSP。

```
Administrator@P-V-9 D:\repository\rt-thread\bsp\stm32f10x-HAL
> scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\applications\main.o
CC build\drivers\board.o
CC build\drivers\drv_gpio.o
CC build\drivers\drv_usart.o
AS build\drivers\gcc_startup.o
CC build\drivers\stm32f1xx_it.o
CC build\kernel\components\drivers\misc\pin.o
CC build\kernel\components\drivers\serial\serial.o
CC build\kernel\components\drivers\src\completion.o
CC build\kernel\components\drivers\src\dataqueue.o
CC build\kernel\components\drivers\src\pipe.o
CC build\kernel\components\drivers\src\ringbuffer.o
CC build\kernel\components\drivers\src\waitqueue.o
CC build\kernel\components\drivers\src\workqueue.o
CC build\kernel\components\finsh\cmd.o
CC build\kernel\components\finsh\msh.o
CC build\kernel\components\finsh\msh_cmd.o
CC build\kernel\components\finsh\msh_file.o
CC build\kernel\components\finsh\shell.o
CC build\kernel\components\finsh\symbol.o
CC build\kernel\components\libc\compilers\newlib\libc.o
CC build\kernel\components\libc\compilers\newlib\libc_syms.o
CC build\kernel\components\libc\compilers\newlib\stdio.o
CC build\kernel\components\libc\compilers\newlib\syscalls.o
```

图 31.1: 使用 scons 命令编译 stm32f10x-HAL BSP

如果用户要使用其他的 BSP 已经支持的编译器编译工程，或者 BSP 为非 ARM 平台的芯片，那么不能直接使用 scons 命令编译工程，需要自己安装对应的编译器，并且指定使用的编译器路径。在编译工程前，可以在 Env 命令行界面使用下面的 2 个命令指定编译器为 MDK 和编译器路径为 MDK 的安装路径。

```
set RTT_CC=keil
set RTT_EXEC_PATH=C:/Keilv5
```

31.2.1 SCons 基本命令

本节介绍 RT-Thread 中常用的 SCons 命令。SCons 不仅完成基本的编译，还可以生成 MDK/IAR/VS 工程。

31.2.1.1 scons

在 Env 命令行窗口进入要编译的 BSP 工程目录，然后使用此命令可以直接编译工程。如果执行过 `scons` 命令后修改了一些源文件，再次执行 `scons` 命令时，则 SCons 会进行增量编译，仅编译修改过的源文件并链接。

如果在 Windows 上执行 `scons` 输出以下的警告信息：

```
scons: warning: No version of Visual Studio compiler found - C/C++ compilers most likely not set correctly.
```

说明 `scons` 并没在你的机器上找到 Visual Studio 编译器，但实际上我们主要是针对设备开发，和 Windows 本地没什么关系，请直接忽略掉它。

`scons` 命令后面还可以增加一个 `-s` 参数，即命令 `scons -s`，和 `scons` 命令不同的是此命令不会打印具体的内部命令。

31.2.1.2 scons -c

清除编译目标。这个命令会清除执行 `scons` 时生成的临时文件和目标文件。

31.2.1.3 scons --target=XXX

如果使用 mdk/iar 来进行项目开发，当修改了 `rtconfig.h` 打开或者关闭某些组件时，需要使用以下命令中的其中一种重新生成对应的定制化的工程，然后在 mdk/iar 进行编译下载。

```
scons --target=iar
scons --target=mdk4
scons --target=mdk5
```

在命令行窗口进入要编译的 BSP 工程目录，使用 `scons --target=mdk5` 命令后会在 BSP 目录生成一个新的 MDK 工程文件名为 `project.uvprojx`。双击它打开，就可以使用 MDK 来编译、调试。使用 `scons --target=iar` 命令后则会生成一个新的 IAR 工程文件名为 `project.eww`。不习惯 SCons 的用户可以使用这种方式。如果打开 `project.uvproj` 失败，请删除 `project.uvopt` 后，重新生成工程。

在 `bsp/simulator` 下，可以使用下面的命令生成 `vs2012` 的工程或 `vs2005` 的工程。

```
scons --target=vs2012
Scons --target=vs2005
```

如果 BSP 目录下提供其他 IDE 工程的模板文件也可以使用此命令生成对应的新工程，比如 `ua`、`vs`、`cb`、`cdk`。

这个命令后面同样可以增加一个 `-s` 参数，如命令 `scons -target=mdk5 -s`，执行此命令时不会打印具体的内部命令。

!!! note “注意事项” 要生成 MDK 或者 IAR 的工程文件，前提条件是 BSP 目录存在一个工程模版文件，然后 scons 才会根据这份模版文件加入相关的源码，头文件搜索路径，编译参数，链接参数等。而至于这个工程是针对哪颗芯片的，则直接由这份工程模版文件指定。所以大多数情况下，这个模版文件是一份空的工程文件，用于辅助 SCons 生成 project.uvprojx 或者 project.eww。

31.2.1.4 scons -jN

多线程编译目标，在多核计算机上可以使用此命令加快编译速度。一般来说一颗 cpu 核心可以支持 2 个线程。双核机器上使用 scons -j4 命令即可。

!!! note “注意事项” 如果你只是想看看编译错误或警告，最好是不使用 -j 参数，这样错误信息不会因为多个文件并行编译而导致出错信息夹杂在一起。

31.2.1.5 scons -dist

搭建项目框架，使用此命令会在 BSP 目录下生成 dist 目录，这便是开发项目的目录结构，包含了 RT-Thread 源码及 BSP 相关工程，不相关的 BSP 文件夹及 libcpu 都会被移除，并且可以随意拷贝此工程到任何目录下使用。

31.2.1.6 scons -verbose

默认情况下，使用 scons 命令编译的输出不会显示编译参数，如下所示：

```
D:\repository\rt-thread\bsp\stm32f10x>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\applications\application.o
CC build\applications\startup.o
CC build\components\drivers\serial\serial.o
...
```

使用 scons -verbose 命令的效果如下：

```
armcc -o build\src\mempool.o -c --device DARMSTM --apcs=interwork -ID:/Keil/ARM/
RV31/INC -g -O0 -DUSE_STDPERIPH_DRIVER -DSTM32F10X_HD -Iapplications -IF:\Projec
t\git\rt-thread\applications -I. -IF:\Project\git\rt-thread -Idrivers -IF:\Proje
ct\git\rt-thread\drivers -ILibraries\STM32F10x_StdPeriph_Driver\inc -IF:\Project
\git\rt-thread\ILibraries\STM32F10x_StdPeriph_Driver\inc -ILibraries\STM32_USB-FS
-Device_Driver\inc -IF:\Project\git\rt-thread\ILibraries\STM32_USB-FS-Device_Driv
er\inc -ILibraries\CMSIS\CMS\DeviceSupport\ST\STM32F10x -IF:\Project\git\rt-thre
...
```

31.3 SCons 进阶

SCons 使用 SConscript 和 SConstruct 文件来组织源码结构，通常来说一个项目只有一 SConstruct，但是会有多个 SConscript。一般情况下，每个存放有源代码的子目录下都会放置一个 SConscript。

为了使 RT-Thread 更好的支持多种编译器，以及方便的调整编译参数，RT-Thread 为每个 BSP 单独创建了一个名为 rtconfig.py 的文件。因此每一个 RT-Thread BSP 目录下都会存在下面三个文件：rtconfig.py、SConstruct 和 SConscript，它们控制 BSP 的编译。一个 BSP 中只有一个 SConstruct 文件，但是却会有多个 SConscript 文件，可以说 SConscript 文件是组织源码的主力军。

RT-Thread 大部分源码文件夹下也存在 SConscript 文件，这些文件会被 BSP 目录下的 SConscript 文件“找到”从而将 rtconfig.h 中定义的宏对应的源代码加入到编译器中来。后文将以 stm32f10x-HAL BSP 为例，讲解 SCons 是如何构建工程。

31.3.1 SCons 内置函数

如果想要将自己的一些源代码加入到 SCons 编译环境中，一般可以创建或修改已有 SConscript 文件。SConscript 文件可以控制源码文件的加入，并且可以指定文件的 Group（与 MDK/IAR 等 IDE 中的 Group 的概念类似）。

SCons 提供了很多内置函数可以帮助我们快速添加源码程序，利用这些函数，再配合一些简单的 Python 语句我们就能随心所欲向项目中添加或者删除源码。下面将简单介绍一些常用函数。

31.3.1.1 GetCurrentDir()

获取当前路径。

31.3.1.2 Glob("*.c")

获取当前目录下的所有 C 文件。修改参数的值为其他后缀就可以匹配当前目录下的所有某类型的文件。

31.3.1.3 GetDepend(macro)

该函数定义在 tools 目录下的脚本文件中，它会从 rtconfig.h 文件读取配置信息，其参数为 rtconfig.h 中的宏名。如果 rtconfig.h 打开了某个宏，则这个方法（函数）返回真，否则返回假。

31.3.1.4 Split(str)

将字符串 str 分割成一个列表 list。

31.3.1.5 DefineGroup(name, src, depend, **parameters)

这是 RT-Thread 基于 SCons 扩展的一个方法（函数）。DefineGroup 用于定义一个组件。组件可以是一个目录（下的文件或子目录），也是后续一些 IDE 工程文件中的一个 Group 或文件夹。

DefineGroup() 函数的参数描述：

参数	描述
name	Group 的名字
src	Group 中包含的文件，一般指的是 C/C++ 源文件。方便起见，也能够通过 Glob 函数采用通配符的方式列出 SConscript 文件所在目录中匹配的文件
depend	Group 编译时所依赖的选项（例如 FinSH 组件依赖于 RT_USING_FINSH 宏定义）。编译选项一般指 rtconfig.h 中定义的 RT_USING_xxx 宏。当在 rtconfig.h 配置文件中定义了相应宏时，那么这个 Group 才会被加入到编译环境中进行编译。如果依赖的宏并没在 rtconfig.h 中被定义，那么这个 Group 将不会被加入编译。相类似的，在使用 scons 生成为 IDE 工程文件时，如果依赖的宏未被定义，相应的 Group 也不会在工程文件中出现
parameters	配置其他参数，可取值见下表，实际使用时不需要配置所有参数

parameters 可加入的参数：

参数	描述
CCFLAGS	C 源文件编译参数
CPPPATH	头文件路径
CPPDEFINES	链接时参数
LIBRARY	包含此参数，则会将组件生成的目标文件打包成库文件

31.3.1.6 SConscript(dirs, variant_dir, duplicate)

读取新的 SConscript 文件，SConscript() 函数的参数描述如下所示：

参数	描述
dirs	SConscript 文件路径
variant_dir	指定生成的目标文件的存放路径
duiplicate	设定是否拷贝或链接源文件到 variant_dir

31.4 SConscript 示例

下面我们将以几个 SConscript 为例讲解 scons 构建工具的使用方法。

31.4.1 SConscript 示例 1

我们先从 stm32f10x-HAL BSP 目录下的 SConscript 文件开始讲解，这个文件管理 BSP 下面的所有其他 SConscript 文件，内容如下所示。

```

import os
cwd = str(Dir('#'))
objs = []
list = os.listdir(cwd)
for d in list:
    path = os.path.join(cwd, d)
    if os.path.isfile(os.path.join(path, 'SConscript')):
        objs = objs + SConscript(os.path.join(d, 'SConscript'))
Return('objs')

```

- `import os`: 导入 Python 系统编程 `os` 模块，可以调用 `os` 模块提供的函数用于处理文件和目录。
- `cwd = str(Dir('#'))`: 获取工程的顶级目录并赋值给字符串变量 `cwd`，也就是工程的 `SConstruct` 所在的目录，在这里它的效果与 `cwd = GetCurrentDir()` 相同。
- `objs = []`: 定义了一个空的 `list` 型变量 `objs`。
- `list = os.listdir(cwd)`: 得到当前目录下的所有子目录，并保存到变量 `list` 中。
- 随后是一个 python 的 `for` 循环，这个 `for` 循环会遍历一遍 BSP 的所有子目录并运行这些子目录的 `SConscript` 文件。具体操作是取出一个当前目录的子目录，利用 `os.path.join(cwd, d)` 拼接成一个完整路径，然后判断这个子目录是否存在一个名为 `SConscript` 的文件，若存在则执行 `objs = objs + SConscript(os.path.join(d, 'SConscript'))`。这一句中使用了 `SCons` 提供的一个内置函数 `SConscript()`，它可以读入一个新的 `SConscript` 文件，并将 `SConscript` 文件中所指明的源码加入到了源码编译列表 `objs` 中来。

通过这个 `SConscript` 文件，BSP 工程所需要的源代码就被加入了编译列表中。

31.4.2 SConscript 示例 2

那么 `stm32f10x-HAL` BSP 其他的 `SConscript` 文件又是怎样的呢？我们再看一下 `drivers` 目录下 `SConscript` 文件，这个文件将管理 `drivers` 目录下面的源代码。`drivers` 目录用于存放根据 RT-Thread 提供的驱动框架实现的底层驱动代码。

```

Import('rtconfig')
from building import *

cwd = GetCurrentDir()

# add the general drivers.
src = Split("""
board.c
stm32f1xx_it.c
""")

if GetDepend(['RT_USING_PIN']):
    src += ['drv_gpio.c']
if GetDepend(['RT_USING_SERIAL']):

```

```

src += ['drv_usart.c']
if GetDepend(['RT_USING_SPI']):
    src += ['drv_spi.c']
if GetDepend(['RT_USING_USB_DEVICE']):
    src += ['drv_usb.c']
if GetDepend(['RT_USING_SDCARD']):
    src += ['drv_sdcard.c']

if rtconfig.CROSS_TOOL == 'gcc':
    src += ['gcc_startup.s']

CPPPATH = [cwd]

group = DefineGroup('Drivers', src, depend = [''], CPPPATH = CPPPATH)

Return('group')

```

- `Import('rtconfig')`: 导入 rtconfig 对象, 后面用到的 `rtconfig.CROSS_TOOL` 定义在这个 `rtconfig` 模块。
- `from building import *`: 把 `building` 模块的所有内容全都导入到当前模块, 后面用到的 `DefineGroup` 定义在这个模块。
- `cwd = GetCurrentDir()`: 获得当前路径并保存到字符串变量 `cwd` 中。

后面一行使用 `Split()` 函数来将一个文件字符串分割成一个列表, 其效果等价于

```
src = ['board.c', 'stm32f1xx_it.c']
```

后面使用了 `if` 判断和 `GetDepend()` 检查 `rtconfig.h` 中的某个宏是否打开, 如果打开, 则使用 `src += [src_name]` 来往列表变量 `src` 中追加源代码文件。

- `CPPPATH = [cwd]`: 将当前路径保存到一个列表变量 `CPPPATH` 中。

最后一行使用 `DefineGroup` 创建一个名为 `Drivers` 的组, 这个组也就对应 MDK 或者 IAR 中的分组。这个组的源代码文件为 `src` 指定的文件, `depend` 为空表示该组不依赖任何 `rtconfig.h` 的宏。

`CPPPATH =CPPPATH` 表示将当前路径添加到系统的头文件路径中。左边的 `CPPPATH` 是 `DefineGroup` 中内置参数, 表示头文件路径。右边的 `CPPPATH` 是本文件上面一行定义的。这样我们就可以在其他源码中引用 `drivers` 目录下的头文件了。

31.4.3 SConscript 示例 3

我们再看一下 `applications` 目录下的 `SConscript` 文件, 这个文件将管理 `applications` 目录下面的源代码, 用于存放用户自己的应用代码。

```

from building import *

cwd = GetCurrentDir()

```

```

src = Glob('*.*c')
CPPPATH = [cwd, str(Dir('#'))]

group = DefineGroup('Applications', src, depend = [''], CPPPATH = CPPPATH)

Return('group')

```

`src = Glob('*.*c')`: 得到当前目录下所有的 C 文件。

`CPPPATH = [cwd, str(Dir('#'))]`: 将当前路径和工程的 SConstruct 所在的路径保存到列表变量 `CPPPATH` 中。

最后一行使用 `DefineGroup` 创建一个名为 `Applications` 的组。这个组的源代码文件为 `src` 指定的文件，`depend` 为空表示该组不依赖任何 `rtconfig.h` 的宏，并将 `CPPPATH` 保存的路径添加到了系统头文件搜索路径中。这样 `applications` 目录和 `stm32f10x-HAL` BSP 目录里面的头文件在源代码的其他地方就可以引用了。

总结：这个源程序会将当前目录下的所有 c 程序加入到组 `Applications` 中，因此如果在这个目录下增加或者删除文件，就可以将文件加入工程或者从工程中删除。它适用于批量添加源码文件。

31.4.4 SConscript 示例 4

下面是 RT-Thread 源代码 component/finsh/SConscript 文件的内容，这个文件将管理 `finsh` 目录下面的源代码。

```

Import('rtconfig')
from building import *

cwd      = GetCurrentDir()
src      = Split('''
shell.c
symbol.c
cmd.c
''')

fsh_src = Split('''
finsh_compiler.c
finsh_error.c
finsh_heap.c
finsh_init.c
finsh_node.c
finsh_ops.c
finsh_parser.c
finsh_var.c
finsh_vm.c
finsh_token.c
'''')

msh_src = Split('''
msh.c
''')

```

```

msh_cmd.c
msh_file.c
''')

CPPPATH = [cwd]
if rtconfig.CROSS_TOOL == 'keil':
    LINKFLAGS = '--keep *.o(FSymTab)'

    if not GetDepend('FINSH_USING_MSH_ONLY'):
        LINKFLAGS = LINKFLAGS + '--keep *.o(VSymTab)'

else:
    LINKFLAGS = ''

if GetDepend('FINSH_USING_MSH'):
    src = src + msh_src
if not GetDepend('FINSH_USING_MSH_ONLY'):
    src = src + fsh_src

group = DefineGroup('finsh', src, depend = ['RT_USING_FINSH'], CPPPATH = CPPPATH,
    LINKFLAGS = LINKFLAGS)

Return('group')

```

我们来看一下文件中第一个 Python 条件判断语句的内容，如果编译工具是 keil，则变量 `LINKFLAGS = '--keep *.o(FSymTab)'` 否则置空。

`DefinGroup` 同样将 `finsh` 目录下的 `src` 指定的文件创建为 `finsh` 组。`depend = ['RT_USING_FINSH']` 表示这个组依赖 `rtconfig.h` 中的宏 `RT_USING_FINSH`。当 `rtconfig.h` 中打开宏 `RT_USING_FINSH` 时，`finsh` 组内的源码才会被实际编译，否则 `SCons` 不会编译。

然后将 `finsh` 目录加入到系统头文件目录中，这样我们就可以在其他源码中引用 `finsh` 目录下的头文件。

`LINKFLAGS = LINKFLAGS` 的含义与 `CPPPATH = CPPPATH` 类似。左边的 `LINKFLAGS` 表示链接参数，右边的 `LINKFLAGS` 则是前面 `if else` 语句所定义的值。也就是给工程指定链接参数。

31.5 使用 SCons 管理工程

前面小节对 RT-Thread 源代码的相关 `SConscript` 做了详细讲解，大家也应该知道了 `SConscript` 文件的一些常见写法，本小节将指导大家如何使用 `SCons` 管理自己的工程。

31.5.1 添加应用代码

前文提到过 `BSP` 下的 `Applications` 文件夹用于存放用户自己的应用代码，目前只有一个 `main.c` 文件。如果用户的应用代码不是很多，建议相关源文件都放在这个文件夹下面。在 `Applications` 文件夹下新增了 2 个简单的文件 `hello.c` 和 `hello.h`，内容如下所示。

```
/* file: hello.h */
```

```

#ifndef _HELLO_H_
#define _HELLO_H_

int hello_world(void);

#endif /* _HELLO_H_ */

/* file: hello.c */
#include <stdio.h>
#include <finsh.h>
#include <rtthread.h>

int hello_world(void)
{
    rt_kprintf("Hello, world!\n");

    return 0;
}

MSH_CMD_EXPORT(hello_world, Hello world!)

```

applications 目录下的 SConscript 文件会把当前目录下的所有源文件都添加到工程中。需要使用 `scons --target=xxx` 命令才会把新增的 2 个文件添加到工程项目中。注意每次新增文件都要重新生成工程。

31.5.2 添加模块

前文提到在自己源代码文件不多的情况下，建议所有源代码文件都放在 applications 文件夹里面。如果用户源代码很多了，并且想创建自己的工程模块，或者需要使用自己获取的其他模块，怎么做会比较合适呢？

同样以上文提到的 `hello.c` 和 `hello.h` 为例，这两个文件将会放到一个单独的文件夹里管理，并且在 MDK 工程文件里有自己的分组，且可以通过 `menuconfig` 选择是否使用这个模块。在 BSP 下新增 `hello` 文件夹。

本地磁盘 (D:) > repository > rt-thread > bsp > stm32f10x-HAL > hello

名称	修改日期	类型	大小
hello.c	2018/8/1 17:43	C 文件	1 KB
hello.h	2018/8/1 17:31	H 文件	1 KB
SConscript	2018/8/4 10:24	文件	1 KB

图 31.2: 新增 `hello` 文件夹

大家注意到文件夹里多了一个 `SConscript` 文件，如果想要将自己的一些源代码加入到 SCons 编译环境中，一般可以创建或修改已有的 `SConscript` 文件。参考上文对 RT-Thread 源代码的一些对 `SConscript` 文件的分析，这个新增的 `hello` 模块 `SConscript` 文件内容如下所示：

```

from building import *

cwd          = GetCurrentDir()
include_path = [ cwd ]
src          = []

if GetDepend(['RT_USING_HELLO']):
    src += ['hello.c']

group = DefineGroup('hello', src, depend = [''], CPPPATH = include_path)

Return('group')

```

通过上面几行简单的代码，就创建了一个新组 hello，并且可以通过宏定义控制要加入到组里面的源文件，还将这个组所在的目录添加到了系统头文件路径中。那么自定义宏 RT_USING_HELLO 又是通过怎样的方式定义呢？这里要介绍一个新的文件 Kconfig。Kconfig 用来配置内核，使用 Env 配置系统时使用的 menuconfig 命令生成的配置界面就依赖 Kconfig 文件。menuconfig 命令通过读取工程的各个 Kconfig 文件，生成配置界面供用户配置内核，最后所有配置相关的宏定义都会自动保存到 BSP 目录里的 rtconfig.h 文件中，每一个 BSP 都有一个 rtconfig.h 文件，也就是这个 BSP 的配置信息。

在 stm32f10x-HAL BSP 目录下已经有了关于这个 BSP 的 Kconfig 文件，我们可以基于这个文件添加自己需要的配置选项。关于 hello 模块添加了如下配置选项，# 号后面为注释。

```

menu "hello module" ..... # 菜单 hello module

... config RT_USING_HELLO ..... # RT_USING_HELLO配置选项
...     bool "Enable hello module" ..... # RT_USING_HELLO类型为 bool, 选中值为 y, 不选中值为 n
...     default y ..... # 默认选中
...     help ..... # 菜单选中 help 后显示的提示信息, 解释配置选项的含义
...     this hello module only used for test

... config RT_HELLO_NAME ..... # RT_HELLO_NAME配置选项
...     string "hello name" ..... # RT_HELLO_NAME类型为字符串
...     default "hello" ..... # 默认值为"hello"

... config RT_HELLO_VALUE ..... # RT_HELLO_VALUE配置选项
...     int "hello value" ..... # RT_HELLO_VALUE类型为 int
...     default 8 ..... # 默认值为8

endmenu

```

图 31.3: hello 模块相关配置选项

使用 Env 工具进入 stm32f10x-HAL BSP 目录后，使用 menuconfig 命令在主页面最下面就可以看到新增的 hello 模块的配置菜单，进入菜单后如下图所示。

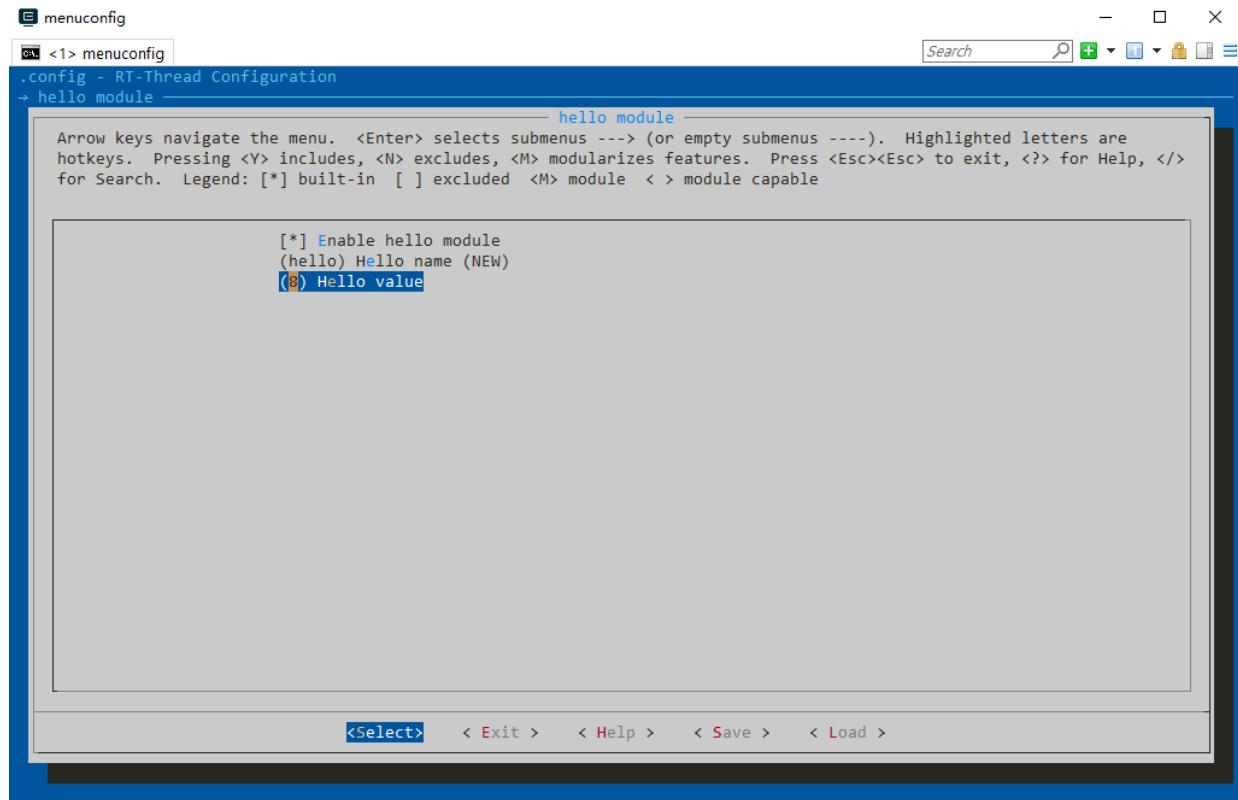


图 31.4: hello 模块配置菜单

还可以修改 hello value 的值。

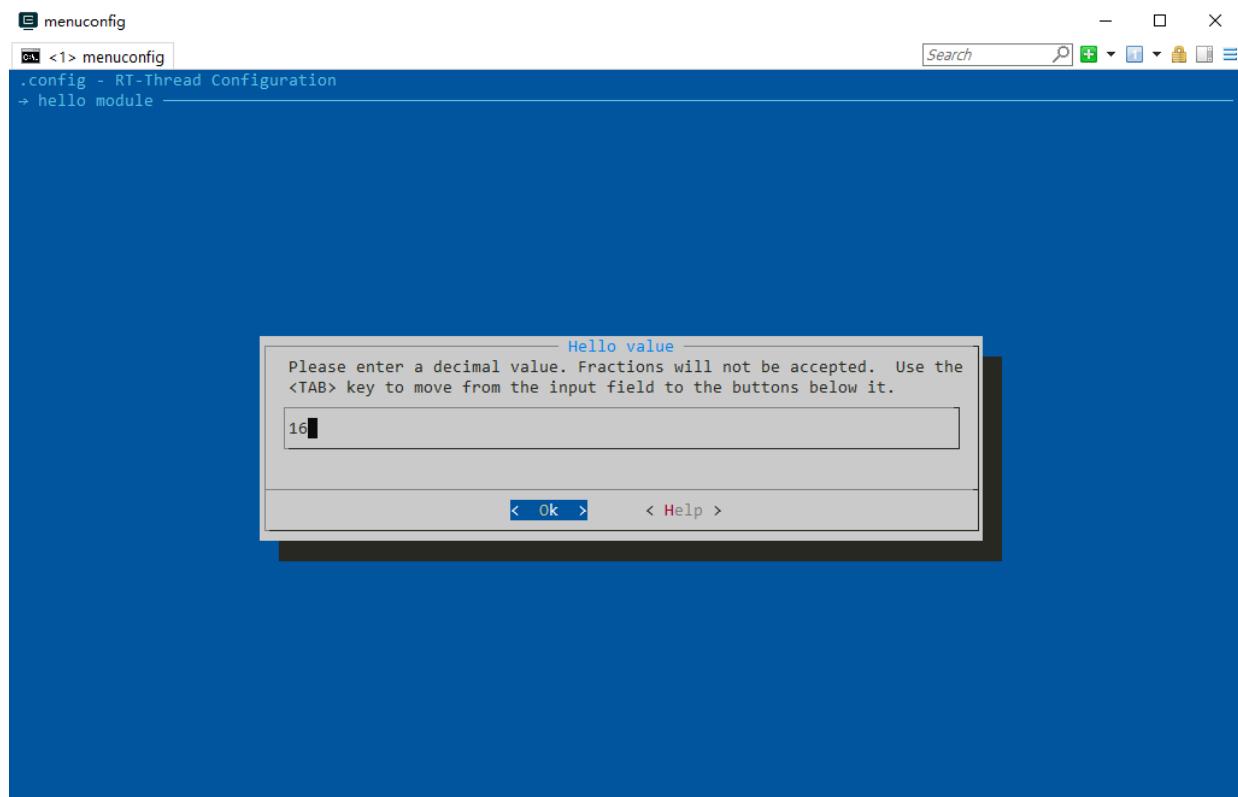
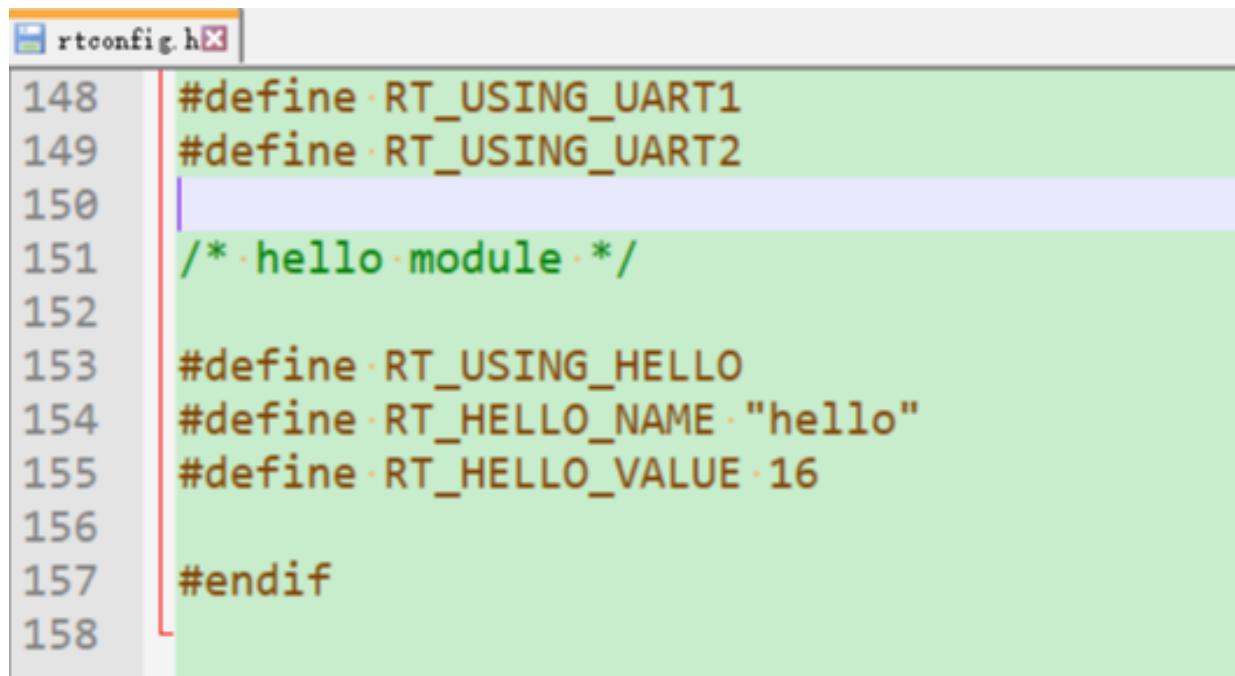


图 31.5: 修改 hello value 的值

保存配置后退出配置界面，打开 stm32f10x-HAL BSP 目录下的 rtconfig.h 文件可以看到 hello 模块的配置信息已经有了。



```

148 #define RT_USING_UART1
149 #define RT_USING_UART2
150
151 /* hello module */
152
153 #define RT_USING_HELLO
154 #define RT_HELLO_NAME "hello"
155 #define RT_HELLO_VALUE 16
156
157 #endif
158

```

图 31.6: hello 模块相关宏定义

注意：每次 **menuconfig** 配置完成后都要使用 **scons -target=XXX** 命令生成新工程。

因为 rtconfig.h 中已经定义了 RT_USING_HELLO 宏，所以新生成工程时就会把 hello.c 的源文件添加到新工程中。

上面只是简单列举了在 Kconfig 文件中添加自己模块的配置选项，用户还可以参考《Env 用户手册》，里面也有对配置选项修改和添加的讲解，也可以自己百度查看 Kconfig 的相关文档，实现其他更复杂的配置选项。

31.5.3 添加库

如果要往工程中添加一个额外的库，需要注意不同的工具链对二进制库的命名。

- ARMCC 工具链下的库名称应该是 **xxx.lib**，一个以.lib 为后缀的文件。
- IAR 工具链下的库名称应该是 **xxx.a**，一个以.a 为后缀的文件。
- GCC 工具链下的库名称应该是 **libxxx.a**，一个以.a 为后缀的文件，并且有 lib 前缀。

ARMCC / IAR 工具链下，若添加库名为 libabc.lib / libabc_iar.a 时，在指定库时指定全名 libabc。

GCC 工具链比较特殊，它识别的是 libxxx.a 这样的库名称，若添加库名为 libabc.a 时，在指定库时是指定 abc，而不是 libabc。

例如，/libs 下有以下库文件需要添加：

```

libabc_keil.lib
libabc_iar.a
libabc_gcc.a

```

则对应的 SConscript 如下：

```
Import('rtconfig')
from building import *

cwd = GetCurrentDir()
src = Split('''
''')

LIBPATH = [ cwd + '/libs' ]          # LIBPATH 指定库的路径，表示库的搜索路径是当前
                                     # 目录下的 'libs' 目录

if rtconfig.CROSS_TOOL == 'gcc':
    LIBS = ['abc_gcc']                # GCC 下 LIBS 指定库的名称
elif rtconfig.CROSS_TOOL == 'keil':
    LIBS = ['libabc_keil']            # ARMCC 下 LIBS 指定库的名称
else:
    LIBS = ['libabc_iar']             # IAR 下 LIBS 指定库的名称

group = DefineGroup('ABC', src, depend = [''], LIBS = LIBS, LIBPATH=LIBPATH)

Return('group')
```

31.5.4 编译器选项

`rtconfig.py` 是一个 RT-Thread 标准的编译器配置文件，控制了大部分编译选项，是一个使用 python 语言编写的脚本文件，主要用于完成以下工作：

- 指定编译器（从支持的多个编译器中选择一个你现在使用的编译器）。
- 指定编译器参数，如编译选项、链接选项等。

当我们使用 `scons` 命令编译工程时，就会按照 `rtconfig.py` 的编译器配置选项编译工程。下面的代码为 `stm32f10x-HAL BSP` 目录下 `rtconfig.py` 的部分代码。

```
import os

# toolchains options
ARCH='arm'
CPU='cortex-m3'
CROSS_TOOL='gcc'

if os.getenv('RTT_CC'):
    CROSS_TOOL = os.getenv('RTT_CC')

# cross_tool provides the cross compiler
# EXEC_PATH is the compiler execute path, for example, CodeSourcery, Keil MDK, IAR

if CROSS_TOOL == 'gcc':
```

```

PLATFORM      = 'gcc'
EXEC_PATH    = '/usr/local/gcc-arm-none-eabi-5_4-2016q3/bin/'
elif CROSS_TOOL == 'keil':
    PLATFORM      = 'armcc'
    EXEC_PATH    = 'C:/Keilv5'
elif CROSS_TOOL == 'iar':
    PLATFORM      = 'iar'
    EXEC_PATH    = 'C:/Program Files/IAR Systems/IAR Workbench 6.0 Evaluation'

if os.getenv('RTT_EXEC_PATH'):
    EXEC_PATH = os.getenv('RTT_EXEC_PATH')

BUILD = 'debug'

if PLATFORM == 'gcc':
    # toolchains
    PREFIX = 'arm-none-eabi-'
    CC = PREFIX + 'gcc'
    AS = PREFIX + 'gcc'
    AR = PREFIX + 'ar'
    LINK = PREFIX + 'gcc'
    TARGET_EXT = 'elf'
    SIZE = PREFIX + 'size'
    OBJDUMP = PREFIX + 'objdump'
    OBJCPY = PREFIX + 'objcopy'

    DEVICE = '-mcpu=cortex-m3 -mthumb -ffunction-sections -fdata-sections'
    CFLAGS = DEVICE
    AFLAGS = '-c' + DEVICE + '-x assembler-with-cpp'
    LFLAGS = DEVICE + '-Wl,--gc-sections,-Map=rtthread-stm32.map,-cref,-u,
                  Reset_Handler -T stm32_rom.ld'

```

其中 CFLAGS 是 C 文件的编译选项, AFLAGS 则是汇编文件的编译选项, LFLAGS 是链接选项。BUILD 变量控制代码优化的级别。默认 BUILD 变量取值为'debug', 即使用 debug 方式编译, 优化级别 0。如果将这个变量修改为其他值, 就会使用优化级别 2 编译。下面几种都是可行的写法(总之只要不是'debug'就可以了)。

```

BUILD = ''
BUILD = 'release'
BUILD = 'hello, world'

```

建议在开发阶段都使用 debug 方式编译, 不开优化, 等产品稳定之后再考虑优化。

关于这些选项的具体含义需要参考编译器手册, 如上面使用的 armcc 是 MDK 的底层编译器。其编译选项的含义在 MDK help 中有详细说明。

前文提到过如果用户执行 scons 命令时希望使用其他编译器编译工程, 可以在 Env 的命令行端使用相关命令指定编译器和编译器路径。但是这样修改只对当前的 Env 进程有效, 再次打开时又需要重新使用命令设置, 我们可以直接修改 rtconfig.py 文件达到永久配置编译器的目的。一般来说, 我们只需要修改 CROSS_TOOL 和下面的 EXEC_PATH 两个选项。

- **CROSS_TOOL:** 指定编译器。可选的值有 keil、gcc、iar，浏览 `rtconfig.py` 可以查看当前 BSP 所支持的编译器。如果您的机器上安装了 MDK，那么可以将 `CROSS_TOOL` 修改为 `keil`，则使用 MDK 编译工程。
- **EXEC_PATH:** 编译器的安装路径。这里有两点需要注意：

安装编译器时（如 MDK、GNU GCC、IAR 等），不要安装到带有中文或者空格的路径中。否则，某些解析路径时会出现错误。有些程序默认会安装到 `C:\Program Files` 目录下，中间带有空格。建议安装时选择其他路径，养成良好的开发习惯。

修改 `EXEC_PATH` 时，需要注意路径的格式。在 windows 平台上，默认的路径分割符号是反斜杠 “\”，而这个符号在 C 语言以及 Python 中都是用于转义字符的。所以修改路径时，可以将 “\” 改为 “/”，或者在前面加 `r`（python 特有的语法，表示原始数据）。

假如某编译器安装位置为 `D:\Dir1\Dir2` 下。下面几种是正确的写法：

- `EXEC_PATH = r'D:\Dir1\Dir2'` 注意，字符串前带有 `r`，则可正常使用 “\”。
- `EXEC_PATH = 'D:/Dir1/Dir2'` 注意，改用 “/”，前面没有 `r`。
- `EXEC_PATH = 'D:\\Dir1\\\\Dir2'` 注意，这里使用 “\” 的转义性来转义 “\” 自己。
- 这是错误的写法：`EXEC_PATH = 'D:\Dir1\Dir2'`。

如果 `rtconfig.py` 文件有以下代码，在配置自己的编译器时请将下列代码注释掉。

```
if os.getenv('RTT_CC'):
    CROSS_TOOL = os.getenv('RTT_CC')
...
if os.getenv('RTT_EXEC_PATH'):
    EXEC_PATH = os.getenv('RTT_EXEC_PATH')
```

上面 2 个 if 判断会设置 `CROSS_TOOL` 和 `EXEC_PATH` 为 Env 的默认值。

编译器配置完成之后，我们就可以使用 SCons 来编译 RT-Thread 的 BSP 了。在 BSP 目录打开命令行窗口，执行 `scons` 命令就会启动编译过程。

31.5.5 RT-Thread 辅助编译脚本

在 RT-Thread 源代码的 `tools` 目录下存放有 RT-Thread 自己定义的一些辅助编译的脚本，例如用于自动生成 RT-Thread 针对一些 IDE 集成开发环境的工程文件。其中最主要的是 `building.py` 脚本。

31.5.6 SCons 更多使用

对于复杂、大型的系统，显然不仅仅是一个目录下的几个文件就可以搞定的，很可能是由数个文件夹一级一级组合而成。

在 SCons 中，可以编写 `SConscript` 脚本文件来编译这些相对独立目录中的文件，同时也可以使用 SCons 中的 `Export` 和 `Import` 函数在 `SConstruct` 与 `SConscript` 文件之间共享数据（也就是 Python 中的一个对象数据）。更多 SCons 的使用方法请参考 [SCons 官方文档](#)。